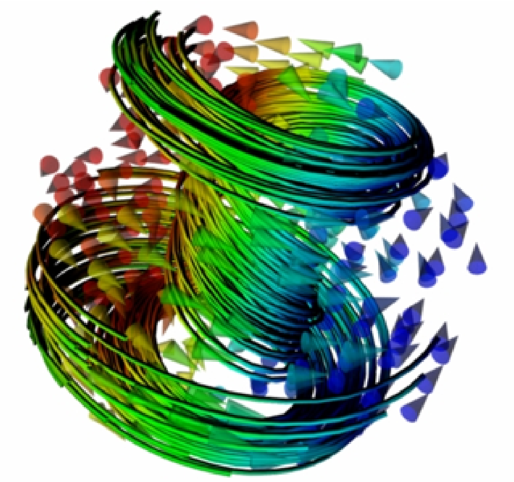
Introduction to

Python for Computational Science and Engineering

(A beginner's guide)



Hans Fangohr
Faculty of Engineering and the Environment
University of Southampton

September 7, 2015

Contents

Chapter 1

Introduction

This text summarises a number of core ideas relevant to Computational Engineering and Scientific Computing using Python. The emphasis is on introducing some basic Python (programming) concepts that are relevant for numerical algorithms. The later chapters touch upon numerical libraries such as `numpy` and `scipy` each of which deserves much more space than provided here. We aim to enable the reader to learn independently how to use other functionality of these libraries using the available documentation (online and through the packages itself).

1.1 Computational Modelling

1.1.1 Introduction

Increasingly, processes and systems are researched or developed through computer simulations: new aircraft prototypes such as for the recent A380 are first designed and tested virtually through computer simulations. With the ever increasing computational power available through supercomputers, clusters of computers and even desktop and laptop machines, this trend is likely to continue.

Computer simulations are routinely used in fundamental research to help understand experimental measurements, and to replace – for example – growth and fabrication of expensive samples/experiments where possible. In an industrial context, product and device design can often be done much more cost effectively if carried out virtually through simulation rather than through building and testing prototypes. This is in particular so in areas where samples are expensive such as nanoscience (where it is expensive to create small things) and aerospace industry (where it is expensive to build large things). There are also situations where certain experiments can only be carried out virtually (ranging from astrophysics to study of effects of large scale nuclear or chemical accidents). Computational modelling, including use of computational tools to post-process, analyse and visualise data, has been used in engineering, physics and chemistry for many decades but is becoming more important due to the cheap availability of computational resources. Computational Modelling is also starting to play a more important role in studies of biological systems, the economy, archeology, medicine, health care, and many other domains.

1.1.2 Computational Modelling

To study a process with a computer simulation we distinguish two steps: the first one is to develop a *model* of the real system. When studying the motion of a small object, such as a penny, say, under the influence of gravity, we may be able to ignore friction of air: our model — which might only consider the gravitational force and the penny's inertia, i.e. $a(t) = F/m = -9.81\mathrm{m/s}^2$ — is an approximation of the real system. The model will normally allow us to express the behaviour of the system (in

some approximated form) through mathematical equations, which often involve ordinary differential equations (ODEs) or partial differential equatons (PDEs).

In the natural sciences such as physics, chemistry and related engineering, it is often not so difficult to find a suitable model, although the resulting equations tend to be very difficult to solve, and can in most cases not be solved analytically at all.

On the other hand, in subjects that are not as well described through a mathematical framework and depend on behaviour of objects whose actions are impossible to predict deterministically (such as humans), it is much more difficult to find a good model to describe reality. As a rule of thumb, in these disciplines the resulting equations are easier to solve, but they are harder to find and the validity of a model needs to be questioned much more. Typical examples are attempts to simulate the economy, the use of global resources, the behaviour of a panicking crowd, etc.

So far, we have just discussed the development of *models* to describe reality, and using these models does not necessarily involve any computers or numerical work at all. In fact, if a model's equation can be solved analytically, then one should do this and write down the solution to the equation.

In practice, hardly any model equations of systems of interest can be solved analytically, and this is where the computer comes in: using numerical methods, we can at least study the model *for a particular set of boundary conditions*. For the example considered above, we may not be able to easily see from a numerical solution that the penny's velocity under the influence of gravity will change linearly with time (which we can read easily from the analytical solution that is available for this simple system: $v(t) = t \cdot 9.81\mathrm{m/s}^2 + v_0$).

The numerical solution that can be computed using a computer would consist of data that shows how the velocity changes over time for a particular initial velocity v_0 (v_0 is a boundary condition here). The computer program would report a long lists of two numbers keeping the (i) value of time t_i for which a particular (ii) value of the velocity v_i has been computed. By plotting all v_i against t_i, or by fitting a curve through the data, we may be able to understand the trend from the data (which we can just see from the analytical solution of course).

It is clearly desirable to find an analytical solutions wherever possible but the number of problems where this is possible is small. Usually, the obtaining numerical result of a computer simulation is very useful (despite the shortcomings of the numerical results in comparison to an analytical expression) because it is the only possible way to study the system at all.

The name *computational modelling* derives from the two steps: (i) *modelling*, i.e. finding a model description of a real system, and (ii) solving the resulting model equations using *computational* methods because this is the only way the equations can be solved at all.

1.1.3 Programming to support computational modelling

A large number of packages exist that provide computational modelling capabilities. If these satisfy the research or design needs, and any data processing and visualisation is appropriately supported through existing tools, one can carry out computational modelling studies without any deeper programming knowledge.

In a research environment – both in academia and research on new products/ideas/... in industry – one often reaches a point where existing packages will not be able to perform a required simulation task, or where more can be learned from analysing existing data in news ways etc.

At that point, programming skills are required. It is also generally useful to have a broad understanding of the building blocks of software and basic ideas of software engineering as we use more and more devices that are software-controlled.

It is often forgotten that there is nothing the computer can do that we as humans cannot do. The computer can do it much faster, though, and also with making far fewer mistakes. There is thus no magic in computations a computer carries out: they could have been done by humans, and – in fact – were for many years (see for example Wikipedia entry on Human Computer).

Understanding how to build a computer simulation comes roughly down to: (i) finding the model (often this means finding the right equations), (ii) knowing how to solve these equations numerically, (ii) to implement the methods to compute these solutions (this is the programming bit).

1.2 Why Python for scientific computing?

The design focus on the Python language is on productivity and code readability, for example through:

- Interactive python console
- Very clear, readable syntax through whitespace indentation
- Strong introspection capabilities
- Full modularity, supporting hierarchical packages
- Exception-based error handling
- Dynamic data types & automatic memory management

As Python is an interpreted language, and it runs many times slower than compiled code, one might ask why anybody should consider such a 'slow' language for computer simulations?

There are two replies to this criticism:

1. *Implementation time versus execution time*: It is not the execution time alone that contributes to the cost of a computational project: one also needs to consider the cost of the development and maintenance work.

   In the early days of scientific computing (say in the 1960/70/80), compute time was so expensive that it made perfect sense to invest many person months of a programmer's time to improve the performance of a calculation by a few percent.

   Nowadays, however, the CPU cycles have become much cheaper than the programmer's time. For research codes which often run only a small number of times (before the researchers move on to the next problem), it may be economic to accept that the code runs only at 25% of the expected possible speed if this saves, say, a month of a researcher's (or programmers) time. For example: if the execution time of the piece of code is 10 hours, and one can predict that it will run about 100 times, then the total execution time is approximately 1000 hours. It would be great if this could be reduced to 25% and one could save 750 (CPU) hours. On the other hand, is an extra wait (about a month) and the cost of 750 CPU hours worth investing one month of a person's time [who could do something else while the calculation is running]? Often, the answer is not.

   Code readability & maintenance - short code, fewer bugs: A related issue is that a research code is not only used for one project, but carries on to be used again and again, evolves, grows, bifurcates etc. In this case, it is often justified to invest more time to make the code fast. At the same time, a significant amount of programmer time will go into (i) introducing the required changes, (ii) testing them even before work on speed optimisation of the changed version can start. To be able to maintain, extend and modify a code in often unforeseen ways, it can only be helpful to use a language that is easy to read and of great expressive power.

2. *Well-written Python code can be very fast* if time critical parts in executed through compiled language.

   Typically, less than 5% percent of the code base of a simulation project need more than 95% of the execution time. As long as these calculations are done very efficiently, one doesn't need to worry about all other parts of the code as the overall time their execution takes is insignificant.

   The compute intense part of the program should to be tuned to reach optimal performance. Python offers a number of options.

   - For example, the `numpy` Python extension provides a Python interface to the compiled and efficient LAPACK libraries that are the quasi-standard in numerical linear algebra. If the problems under study can be formulated such that eventually large systems of algebraic equations have to be solved, or eigenvalues computed, etc, then the compiled code in the LAPACK library can be used (through the Python-numpy package). At this stage, the calculations are carried out with the same performance of Fortran/C as it is essentially Fortran/C code that is used. Matlab, by the way, exploits exactly this: the Matlab scripting language is very slow (about 10 time slower than Python), but Matlab gains its power from delegating the matix operation to the compiled LAPACK libraries.
   - Existing numerical C/Fortran libraries can be interfaced to be usable from within Python (using for example Swig, Boost.Python and Cython).
   - Python can be extended through compiled languages if the computationally demanding part of the problem is algorithmically non-standard and no existing libraries can be used.

     Commonly used are C, Fortran and C++ to implement fast extensions.
   - We list some tools that are used to use compiled code from Python:
     - ▷ The `scipy.weave` extension is useful if just a short expression needs to be expressed in C.
     - ▷ The Cython interface is growing in popularity to (i) semi-automatically declare variable types in Python code, to translate that code to C (automatically) and to then use the compiled C code from Python. Cython is also used to quickly wrap an existing C library with an interface so the C library can be used from Python.
     - ▷ Boost.Python is specialised for wrapping C++ code in Python.

The conclusion is that Python is "fast enough" for most computational tasks, and that its user friendly high-level language often makes up for reduced speed in comparison to compiled lower-level languages. Combining Python with tailor-written compiled code for the performance critical parts of the code, results in virtually optimal speed in most cases.

1.2.1 Optimisation strategies

We generally understand reduction of execution time when discussing "code optimisation" in the context of computational modelling, and we essentially like to carry out the required calculations as fast as possible. (Sometimes we need to reduce the amount of RAM, the amount of data input output to disk or the network.) At the same time, we need to make sure that we do not invest inappropriate amounts of programming time to achieve this speed up: as always there needs to be a balance between the programmers' time and the improvement we can gain from this.

1.2.2 Get it right first, then make it fast

To write fast code effectively, we note that the right order is to (i) first write a program that carries out the correct calculation. For this, choose a language/approach that allows you to *write the code quickly and make it work quickly* — regardless of execution speed. Then (ii) either change the program or re-write it from scratch in the same language to make the execution faster. During the process, keep comparing results with the slow version written first to make sure the optimisation does not introduce errors. (Once we are familiar with the concept of regression tests, they should be used here to compare the new and hopefully faster code with the original code.)

A common pattern in Python is to start writing pure Python code, then start using Python libraries that use compiled code internally (such as the fast arrays Numpy provides, and routines from scipy that go back to established numerical codes such as ODEPACK, LAPACK and others). If required, one can – after careful profiling – start to replace parts of the Python code with a compiled language such as C and Fortran to improve execution speed further (as discussed above).

1.2.3 Prototyping in Python

It turns out that – even if a particular code has to be written in, say, C++ – it is (often) more time efficient to prototype the code in Python, and once an appropriate design (and class structure) has been found, to translate the code to C++.

1.3 Literature

While this text starts with an introduction of (some aspects of) the basic Python programming language, you may find - depending on your prior experience - that you need to refer to secondary sources to fully understand some ideas.

We repeatedly refer to the following documents:

- Allen Downey, *Think Python*. Available online in html and pdf at http://www.greenteapress.com/thinkpython/thinkpython.html, or from Amazon.
- The Python documentation http://www.python.org/doc/, and:
- The Python tutorial (http://docs.python.org/tutorial/)

You may also find the following links useful:

- The `numpy` home page (http://numpy.scipy.org/)
- The `scipy` home page (http://scipy.org/)
- The `matplotlib` home page (http://matplotlib.sourceforge.net/).
- The Python style guide (http://www.python.org/dev/peps/pep-0008/

1.3.1 Recorded video lectures on Python for beginners

Do you like to listen/follow lectures? There is a series of 24 lectures titled *Introduction to Computer Science and Programming* delivered by Eric Grimsom and John Guttag from the MIT available at http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/ This is aimed at students with little or no programming experience. It aims to provide students with an understanding of the role computation can play in solving problems. It also aims to help students, regardless of their major, to feel justifiably confident of their ability to write small programs that allow them to accomplish useful goals.

1.3.2 Python tutor mailing list

There is also a Python tutor mailing list (http://mail.python.org/mailman/listinfo/tutor) where beginners are welcome to ask questions regarding Python. Both using the archives and posting your own queries (or in fact helping others) may help with understanding the language. Use the normal mailing list etiquette (i.e. be polite, concise, etc). You may want to read http://www.catb.org/esr/faqs/smart-questions.html for some guidance on how to ask questions on mailing lists.

1.4 Python version

There are two version of the Python language out there: Python 2.x and Python 3.x. They are (slightly) different — the changes in Python 3.x were introduced to address shortcomings in the design of the language that were identified since Python's inception. A decision has been made that some incompatibilty should be accepted to achieve the higher goal of a better language for the future.

For scientific computation, it is crucial to make use of numerical libraries such as numpy, scipy and the plotting package matplotlib.

All of these are available for Python 2.x, and increasingly they are also available for Python 3 (in fact the libraries above have all been ported by now). As Python 2.x is still the default Python on many system and there are a fair number of research codes out there based on Python 2, we will use Python 2.x in this book.

However, we will write code that is as much as possible in the Python 3 style (and understood by Python 2). The most prominent example is that in Python 2.x, the `print` command is special where as in Python 3 it is an ordinary function. For example, in Python 2.7, we can write

```
print "Hello World"
```

where as in Python 3, this would cause a SyntaxError. The right way to use `print` in Python 3 would be as a function, i.e.

```
print("Hello World")
```

See also section 5.1.5 on page 54 for further details.

Fortunately, the function notation (i.e. with the parantheses) is also allowed in Python 2.7, so we choose this notation in our examples and thus they will execute in Python 2.7 and Python 3.x. (There are other differences.)

The transition of all actively maintained codes from Python 2 to Python 3 is likely to take at least another 5 years, maybe 10. It could also be that Python 2.7 will remain longer actively used – this is hard to predict at the moment.

1.5 This document

This document has been typeset with the `\hyperref` package. This means that all entries in the table of contents, figure numbers, page numbers and URLs should appear as clickable hyperlinks if your pdf browser supports this. You may want to make use of this as often there are URLs provided that provide further information/documentation.

1.6 Your feedback

is desired. If you find anything wrong in this text, or have suggestions how to change or extend it, please feel free to contact Hans at `fangohr@soton.ac.uk`.

If you find a URL that is not working (or pointing to the wrong material), please let Hans know as well. As the content of the Internet is changing rapidly, it is difficult to keep up with these changes without feedback.

Chapter 2

A powerful calculator

2.1 Python prompt and Read-Eval-Print Loop (REPL)

Python is an *interpreted* language. We can either collect sequences of commands into text files and save this to file as a *Python program*. It is convention that these files have the file extension "`.py`", for example `hello.py`.

We can also enter individual commands at the Python prompt which are immediately evaluated and carried out by the Python interpreter. This is very useful for the programmer/learner to understand how to use certain commands (often before one puts these commands together in a longer Python program). Python's role can be described as Reading the command, Evaluating it, Printing the evaluated value and repeating (Loop) the cycle – this is the origin of the REPL abbreviation.

To start the interpreter we can either

- On windows: start IDLE
- On windows: find the MS-DOS prompt, and type `python.exe` followed by the return key.
- On Linux/Unix/Mac OSX: find a shell (called "terminal" in Applications/Utilities on Mac OS X and type `python` followed by the return key.

The python prompt (the chevron `>>>`) signals that Python is waiting for input from us:

```
>>>
```

We can now enter commands, for example `4+5`, followed by the RETURN key.

```
>>> 4+5
9
>>>
```

Once we press the return key, Python will evaluate the expression (`4+5`) and display the computed value (`9`) in the next line. It then displays the python prompt (`>>>`) in the next line to indicate that it is ready for the next input.

This interactive programming environment is sometimes referred to as the read-eval-print loop (REPL), because the expression is read, evaluated, the result is printed, and then the loop starts again.

2.2 Calculator

Basic operations such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and exponentiation (`**`) work (mostly) as expected:

```
>>> 10+10000
10010
>>> 42-1.5
40.5
>>> 47*11
517
>>> 10/0.5
20.0
>>> 2**2
4
>>> 2**3
8
>>> 2**4
16
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
```

and, using the fact that $\sqrt[n]{x} = x^{1/n}$, we can compute the $\sqrt{3} = 1.732050\ldots$ using `**`:

```
>>> 3**0.5
1.7320508075688772
```

Parenthesis can be used for grouping:

```
>>> 2*10+5
25
>>> 2*(10+5)
30
```

2.3 Integer division

Unexpected behaviour can occur when dividing two integer numbers:

```
>>> 15/6
3
```

This phenomenon is known (in many programming languages, including C) as *integer division*: because we provide two integer numbers (15 and 6) to the division operator (/), the assumption that Python makes is that we seek a return value of type integer. The mathematically correct answer is (the floating point number) 2.5. (→ numerical data types in section 3.2.)

The convention for integer division is to truncate the fractional digits and to return the integer part only (i.e. 2 in this example). It is also called “floor division”.

2.3.1 How to avoid integer division

There are two ways to avoid the problem of integer division:

1. Make use of Python's future division: it has been decided that from Python 3.0 onwards the division operator will return a floating point number (complex, if required) even if the numerator and denominator are of integer type. This feature can be activated in older (2.x) Python versions with the `from __future__ import division` statement:

```
>>> 21/7
3
>>> 15/6
2
>>> from __future__ import division
>>> 15/6
2.5
>>> 21/7
3.0
```

   If you want to use the `from __future__ import division` feature in a python program, it would normally be included at the beginning of the file.

2. Alternatively, if we ensoure that at least one number (numerator or denominator) is of type float (or complex), the division operator will return a floating point number. This can be done by writing `15.` instead of `15`, of by forcing conversion of the number to a float, i.e. use `float(15)` instead of `15`:

```
>>> 15/6
2
>>> 15./6
2.5
>>> 15.0/6
2.5
>>> float(15)/6
2.5
>>> 15/6.
2.5
>>> 15/float(6)
2.5
>>> 15./6.
2.5
```

If we really want integer division, we can use `//`: `1//2` returns 0 (true in version 2.x, 3.x and the foreseeable future).

2.3.2 Why should I care about this division problem?

Integer division can result in surprising bugs: suppose you are writing code to compute the mean value $m = (x + y)/2$ of two numbers x and y. The first attempt of writing this may read:

```
m = (x + y) / 2
```

Suppose this is tested with $x = 0.5, y = 0.5$, then the line above computes the correct answers $m = 0.5$ (because`0.5 + 0.5 = 1.0`, i.e. a 1.0 is a floating point number, and thus `1.0/2` evaluates to `0.5`). Or we could use $x = 10, y = 30$, and because `10 + 30 = 40` and `40/2` evaluates to `20`, we get the correct

answer $m = 20$. However, if the integers $x = 0$ and $y = 1$ would come up, then the code returns $m = 0$ (because `0 + 1 = 1` and `1/2` evaluates to `0`) whereas $m = 0.5$ would have been the right answer.

We have many possibilities to change the line of code above to work safely, including these three versions:

```
m = (x + y) / 2.0
```

```
m = float(x + y) / 2
```

```
m = (x + y) * 0.5
```

This integer division behaviour is common amongst most programming languages (including the important ones C, C++ and Fortran), and it is important to be aware of the issue.

2.4 Mathematical functions

Because Python is a general purpose programming language, commonly used mathematical functions such as sin, cos, exp, log and many others are located in the mathematics module with name `math`. We can make use of this as soon as we *import* the math module:

```
>>> import math
>>> math.exp(1.0)
2.7182818284590451
```

Using the `dir` function, we can see the directory of objects available in the math module:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor',
'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

As usual, the `help` function can provide more information about the module (`help(math)`) on individual objects:

```
>>> help(math.exp)
Help on built-in function exp in module math:
exp(...)
    exp(x)
    Return e raised to the power of x.
```

The mathematics module defines to constants π and e:

```
>>> math.pi
3.1415926535897931
>>> math.e
2.7182818284590451
>>> math.cos(math.pi)
-1.0
>>> math.log(math.e)
1.0
```

2.5 Variables

A *variable* can be used to store a certain value or object. In Python, all numbers (and everything else, including functions, modules and files) are objects. A variable is created through assignement:

```
>>> x = 0.5
>>>
```

Once the variable x has been created through assignment of 0.5 in this example, we can make use of it:

```
>>> x*3
0.5
>>> x**2
0.25
>>> y = 111
>>> y+222
333
```

A variable is overriden if a new value is assigned:

```
>>> y = 0.7
>>> math.sin(y) ** 2 + math.cos(y) ** 2
1.0
```

The equal sign ('=') is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # initialise x, y and z with 0
>>> x
0
>>> y
0
>>> z
0
```

Variables must be created (assigned a value) before they can be used, or an error will occur:

```
>>> # try to access an undefined variable
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

2.5.1 Terminology

Strictly speaking, the following happens when we write

```
>>> x = 0.5
```

First, Python creates the object `0.5`. Everything in Python is an object, and so is the floating point number 0.5. This object is stored somewhere in memory. Next, Python *binds a name to the object*. The name is `x`, and we often refer casually to `x` as a variable, an object, or even the value 0.5. However, technically, `x` is a name that is bound to the object `0.5`. Another way to say this is that `x` is a reference to the object.

While it is often sufficient to think about assigning 0.5 to a variable x, there are situations where we need to remember what actually happens. In particular, when we pass references to objects to functions, we need to realise that the function may operate on the object (rather than a copy of the object). This is discussed in more detail in 3.4.

2.6 Impossible equations

In computer programs we often find statements like

```
x = x + 1
```

If we read this as an equation as we are use to from mathematics,

$$x = x + 1$$

we could subtract x on both sides, to find that

$$0 = 1.$$

We know this is not true, so something is wrong here.

The answer is that "equations" in computer codes are not equations but *assignments*. They always have to be read in the following way two-step way:

1. Evaluate the value on the right hand side of the equal sign
2. Assign this value to the variable name shown on the left hand side. (In Pyton: bind the name on the left hand side to the object shown on the right hand side.)

Some computer science literature uses the following notation to express assignments and to avoid the confusion with mathematical equations:

$$x \leftarrow x + 1$$

Let's apply our two-step rule to the assignment `x = x + 1` given above:

1. Evaluate the value on the right hand side of the equal sign: for this we need to know what the current value of `x` is. Let's assume `x` is currently `4`. In that case, the right hand side `x+1` evaluates to `5`.

2. Assign this value (i.e. `5`) to the variable name shown on the left hand side `x`.

Let's confirm with the Python prompt that this is the correct interpretation:

```
>>> x = 4
>>> x = x + 1
>>> print x
5
```

2.6.1 The += notation

Because it is a quite a common operation to increase a variable `x` by some fixed amount `c`, we can write

```
x += c
```

instead of

```
x = x + c
```

Our initial example above could thus have been written

```
>>> x = 4
>>> x += 1
>>> print x
5
```

The same operators are defined for multiplication with a constant (`*=`), subtraction of a constant (`-=`) and division by a constant (`/=`).

Note that the order of `+` and `=` matters:

```
x += 1
```

will increase the variable `x` by one where as

```
x =+ 1
```

will assign the value `+1` to the variable `x`.

Chapter 3

Data Types and Data Structures

3.1 What type is it?

Python knows different data types. To find the type of a variable, use the `type()` function:

```
>>> a = 45
>>> type(a)
<type 'int'>
>>> b = 'This is a string'
>>> type(b)
<type 'str'>
>>> c = 2 + 1j
>>> type(c)
<type 'complex'>
>>> d = [1, 3, 56]
>>> type(d)
<type 'list'>
```

3.2 Numbers

Further information

- Informal introduction to numbers. Python tutorial, section 3.1.1
- Python Library Reference: formal overview of numeric types, http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex
- Think Python, Sec 2.1

The in-built numerical types are integers (see section 3.2.1) and floating point numbers (see section 3.2.3) and complex floating point numbers (section 3.2.4). There are also so-called *long integers* (3.2.2 which have no upper or lower limit (assuming the machine provides enough RAM)

3.2.1 Integers

We have seen the use of integer numbers already in section 2.2. Be aware of integer division problems (section 2.3).

If we need to convert string containing an integer number to an integer we can use `int()` function:

```
>>> a = '34'          # a is a string containing the characters 3 and 4
>>> x = int(a)        # x is in integer number
```

The function `int()` will also convert floating point numbers to integers:

```
>>> int(7.0)
7
>>> int(7.9)
7
```

Note than `int` will truncate any non-integer part of a floating point number. To `round` an floating point number to an integer, use the `round()` command, and then convert the rounded float into an int:

```
>>> round(7.9)
8.0
>>> int(round(7.9))
8
```

3.2.2 Long integers

In line with other programming languages and the support of integer arithmetic through today's CPUs, there is an upper limit for the integers that can be presented. In Python, the `sys` module provides this number in `sys.maxint`:

```
>>> import sys
>>> sys.maxint
2147483647
```

If this range is exceeded, then Python will change the type of the number from `int` to `long`:

```
>>> type(sys.maxint)
<type 'int'>
>>> sys.maxint+1
2147483648L
>>> type(sys.maxint+1)
<type 'long'>
```

The *long integer* data type behaves like an integer but any arithmetic involving long integers is carried out at a software level. This avoids integer overflows but we should note that operations involving long integers are significantly slower than operations with integers. There is no limit to the maximum or minimum long integer than could be used (although the longer the number the more RAM is required and the more CPU time to carry out calculations).

There is no comparable long int type in C or Matlab.

(In Python 3.0, the distinction between int and long int will disappear.)

3.2.3 Floating Point numbers

A string containing a floating point number can be converted into a floating point number using the `float()` command:

```
>>> a = '35.342'
>>> b = float(a)
```

```
>>> print b
35.342
>>> print type(b)
<type 'float'>
```

3.2.4 Complex numbers

Python (as Fortran and Matlab) has built-in complex numbers. Here are some examples how to use these:

```
>>> x = 1 + 3j
>>> x
(1+3j)
>>> abs(x)                #computes the absolute value
3.1622776601683795
>>> x.imag
3.0
>>> x.real
1.0
>>> x * x
(-8+6j)
>>> x * x.conjugate()
(10+0j)
>>> 3 * x
(3+9j)
```

Note that if you want to perform more complicated operations (such as taking the square root, etc) you have to use the `cmath` module (Complex MATHematics):

```
>>> import cmath
>>> cmath.sqrt(x)
(1.4426152744526283+1.0397782600555705j)
```

3.2.5 Functions applicable to all types of numbers

The `abs()` function returns the absolute value of a number (also called modulus):

```
>>> a = -45.463
>>> print abs(a)
45.463
```

Note that `abs()` also works for complex numbers (see 3.2.4).

3.3 Sequences

Strings (3.3.1), lists (3.3.2) and tuples (3.3.3) are *sequences*. They can be *indexed* (3.3.4) and *sliced* (3.3.5) in the same way.

Tuples and strings are "immutable" (which basically means we can't change individual elements within the tuple, and we cannot change individual characters within a string) whereas lists are "mutable" (*.i.e* we can change elements in a list.)

Sequences share the following operations

`a[i]`	returns i-th element of `a`
`a[i:j]`	returns elements i up to $j-1$
`len(a)`	returns number of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`x in a`	returns `True` if `x` is element in `a`
`a + b`	concatenates `a` and `b`
`n * a`	creates `n` copies of sequence `a`

3.3.1 Sequence type 1: String

Further information

- Introduction to strings, Python tutorial 3.1.2

A string is a (immutable) sequence of characters. A string can be defined using single quotes:

```
>>> a = 'Hello World'
```

double quotes:

```
>>> a = "Hello World"
```

or triple quotes of either kind

```
>>> a = """Hello World"""
>>> a = '''Hello World'''
```

The type of a string is `str` and the empty string is given through `""`:

```
>>> a = "Hello World"
>>> type(a)
<type 'str'>
>>> b = ""
>>> type(b)
<type 'str'>
>>> type("Hello World")
<type 'str'>
>>> type("")
<type 'str'>
```

The number of characters in a string (that is its *length*) can be obtained using the `len()`-function:

```
>>> a = "Hello Moon"
>>> len(a)
10
>>> a = 'test'
>>> len(a)
4
>>> len('another test')
12
```

You can combine ("concatenate") two strings using the `+` operator:

```
>>> 'Hello ' + 'World'
'Hello World'
```

Strings have a number of useful methods, including for example `upper()` which returns the string in upper case:

```
>>> a = "This is a test sentence."
>>> a.upper()
'THIS IS A TEST SENTENCE.'
```

A list of available string methods can be found in the Python reference documentation. If a Python prompt is available, one should use the `dir` and `help` function to retrieve this information, *i.e.* `dir("")` provides the list of methods, `help` can be used to learn about each method.

A particularly useful method is `split()` which converts a string into a list of strings:

```
>>> a = "This is a test sentence."
>>> a.split()
['This', 'is', 'a', 'test', 'sentence.']
```

The `split()` method will separate the string where it finds *white space*. White space means any character that is printed as white space, such as one space or several spaces or a tab.

By passing a separator character to the `split()` method, a string can split into different parts. Suppose, for example, we would like to obtain a list of complete sentences:

```
>>> a = "The dog is hungry. The cat is bored. The snake is awake."
>>> a.split(".")
['The dog is hungry', ' The cat is bored', ' The snake is awake', '']
```

The opposite string method to `split` is `join` which can be used as follows:

```
>>> a = "The dog is hungry. The cat is bored. The snake is awake."
>>> s = a.split('.')
>>> s
['The dog is hungry', ' The cat is bored', ' The snake is awake', '']
>>> ".".join(s)
'The dog is hungry. The cat is bored. The snake is awake.'
>>> " STOP".join(s)
'The dog is hungry STOP The cat is bored STOP The snake is awake STOP'
```

3.3.2 Sequence type 2: List

Further information

- Introduction to Lists, Python tutorial, section 3.1.4

A list is a sequence of objects. The objects can be of any type, for example integers:

```
>>> a = [34, 12, 54]
```

or strings:

```
>>> a = ['dog', 'cat', 'mouse']
```

An empty list is presented by `[]`:

```
>>> a = []
```

The type is `list`:

```
>>> type(a)
<type 'list'>
>>> type([])
<type 'list'>
```

As with strings, the number of elements in a list can be obtained using the `len()` function:

```
>>> a = ['dog', 'cat', 'mouse']
>>> len(a)
3
```

It is also possible to *mix* different types in the same list:

```
>>> a = [123, 'duck', -42, 17, 0, 'elephant']
```

In Python a list is an object. It is therefor possible for a list to contain other lists (because a list keeps a sequence of objects):

```
a = [1, 4, 56, [5, 3, 1], 300, 400]
```

You can combine ("concatenate") two lists using the `+` operator:

```
>>> [3, 4, 5] + [34, 35, 100]
[3, 4, 5, 34, 35, 100]
```

Or you can add one object to the end of a list using the `append()` method:

```
>>> a = [34, 56, 23]
>>> a.append(42)
>>> print a
[34, 56, 23, 42]
```

You can delete an object from a list by calling the `remove()` method and passing the object to delete. For example:

```
>>> a = [34, 56, 23, 42]
>>> a.remove(56)
>>> print a
[34, 23, 42]
```

The range() command

A special type of list is frequently required (often together with `for-loops`) and therefor a command exists to generate that list: the `range(n)` command generates a list of integers starting from 0 and going up to *but not including* n. Here are a few examples:

```
>>> range(3)
[0, 1, 2]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This command is often used with for loops. For example, to print the numbers 0^2,1^2,2^2,3^2,…,10^2, the following program can be used:

```
>>> for i in range(11):
...     print i ** 2
...
0
1
4
9
16
25
36
49
64
81
100
```

The range command takes an optional parameter for the beginning of the integer sequence (start) and another optional parameter for the step size. This is often written as `range([start],stop,[step])` where the arguments in square brackets (*i.e.* start and step) are optional. Here are some examples:

```
>>> range(3, 10)            # start=3
[3, 4, 5, 6, 7, 8, 9]
>>> range(3, 10, 2)         # start=3, step=2
[3, 5, 7, 9]
>>> range(10, 0, -1)        # start=10,step=-1
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

3.3.3 Sequence type 3: Tuples

A *tuple* is a (immutable) sequence of objects. Tuples are very similar in behaviour to lists with the exception that they cannot be modified (i.e. are immutable).

For example, the objects in a sequence can be of any type:

```
>>> a = (12, 13, 'dog')
>>> a
(12, 13, 'dog')
>>> a[0]
12
```

The parentheses are not necessary to define a tuple: just a sequence of objects separated by commas is sufficient to define a tuple:

```
>>> a = 100, 200, 'duck'
>>> a
(100, 200, 'duck')
```

although it is good practice to include the paranthesis where it helps to show that tuple is defined.

Tuples can also be used to make two assignments at the same time:

```
>>> x, y = 10, 20
>>> x
```

```
10
>>> y
20
```

This can be used to *swap* to objects within one line. For example

```
>>> x = 1
>>> y = 2
>>> x, y = y, x
>>> print x
2
>>> print y
1
```

The empty tuple is given by `()`

```
>>> t = ()
>>> len(t)
0
>>> type(t)
<type 'tuple'>
```

The notation for a tuple containing one value may seem a bit odd at first:

```
>>> t = (42,)
>>> type(t)
<type 'tuple'>
>>> len(t)
1
```

The extra comma is required to distinguish `(42,)` from `(42)` where in the latter case the parenthesis would be read as defining operator precedence: `(42)` simplifies to `42` which is just a number:

```
>>> t = (42)
>>> type(t)
<type 'int'>
```

This example shows the immutability of a tuple:

```
>>> a = (12, 13, 'dog')
>>> a[0]
12
>>> a[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

The immutability is the main difference between a tuple and a list (the latter being mutable). We should use tuples when we don't want the content to change.

Note that Python functions that return more than one value, return these in tuples (which makes sense because you don't want these values be changed).

3.3.4 Indexing sequences

Further information

- Introduction to strings and indexing in Python tutorial, section 3.1.2, the relevant section is starting after strings have been introduced.

Individual objects in lists can be accessed by using the index of the object and square brackets (`[` and `]`):

```
>>> a = ['dog', 'cat', 'mouse']
>>> a[0]
'dog'
>>> a[1]
'cat'
>>> a[2]
'mouse'
```

Note that Python (like C but unlike Fortran and unlike Matlab) starts counting indices from zero!

Python provides a handy shortcut to retrieve the last element in a list: for this one uses the index "-1" where the minus indicates that it is one element *from the back* of the list. Similarly, the index "-2" will return the 2nd last element:

```
>>> a = ['dog', 'cat', 'mouse']
>>> a[-1]
'mouse'
>>> a[-2]
'cat'
```

If you prefer, you can think of the index `a[-1]` to be a shorthand notation for `a[len(a) - 1]`.

Remember that strings (like lists) are also a sequence type and can be indexed in the same way:

```
>>> a = "Hello World!"
>>> a[0]
'H'
>>> a[1]
'e'
>>> a[10]
'd'
>>> a[-1]
'!'
>>> a[-2]
'd'
```

3.3.5 Slicing sequences

Further information

- Introduction to strings, indexing and slicing in Python tutorial, section 3.1.2

Slicing of sequences can be used to retrieve more than one element. For example:

```
>>> a = "Hello World!"
>>> a[0:3]
'Hel'
```

By writing `a[0:3]` we request the first 3 elements starting from element 0. Similarly:

```
>>> a[1:4]
'ell'
>>> a[0:2]
'He'
>>> a[0:6]
'Hello '
```

We can use negative indices to refer to the end of the sequence:

```
>>> a[0:-1]
'Hello World'
```

It is also possible to leave out the start or the end index and this will return all elements up to the beginning or the end of the sequence. Here are some examples to make this clearer:

```
>>> a = "Hello World!"
>>> a[:5]
'Hello'
>>> a[5:]
' World!'
>>> a[-2:]
'd!'
>>> a[:]
'Hello World!'
```

Note that `a[:]` will generate a *copy* of `a`.

The use of indices in slicing is by some people experienced as counter intuitive. If you feel uncomfortable with slicing, have a look at this quotation from the Python tutorial (section 3.1.2):

> The best way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of 5 characters has index 5, for example:

```
 +---+---+---+---+---+
 | H | e | l | l | o |
 +---+---+---+---+---+
 0   1   2   3   4   5    <-- use for SLICING
-5  -4  -3  -2  -1        <-- use for SLICING
                              from the end
```

> The first row of numbers gives the position of the slicing indices 0...5 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labelled i and j, respectively.

So the important statement is that for *slicing* we should think of indices pointing between characters.

For *indexing* it is better to think of the indices referring to characters. Here is a little graph summarising these rules:

```
   0   1   2   3   4     <-- use for INDEXING
  -5  -4  -3  -2  -1     <-- use for INDEXING
 +---+---+---+---+---+           from the end
 | H | e | l | l | o |
 +---+---+---+---+---+
 0   1   2   3   4   5   <-- use for SLICING
-5  -4  -3  -2  -1       <-- use for SLICING
                                 from the end
```

If you are not sure what the right index is, it is always a good technique to play around with a small example at the Python prompt to test things before or while you write your program.

3.3.6 Dictionaries

Dictionaries are also called "associative arrays" and "hash tables". Dictionaries are *unordered* sets of *key-value pairs*.

An empty dictionary can be created using curly braces:

```
>>> d = {}
```

Keyword-value pairs can be added like this:

```
>>> d['today'] = '22 deg C'     # 'today' is the keyword
                                # '22 deg C' is the value
>>> d['yesterday'] = '19 deg C'
```

`d.keys()` returns a list of all keys:

```
>>> d.keys()
['yesterday', 'today']
```

We can retrieve values by using the keyword as the index:

```
>>> print d['today']
22 deg C
```

Other ways of populating a dictionary if the data is known at creation time are:

```
>>> d2 = {2:4, 3:9, 4:16, 5:25}
>>> d2
{2: 4, 3: 9, 4: 16, 5: 25}
>>> d3 = dict(a=1, b=2, c=3)
>>> d3
{'a': 1, 'c': 3, 'b': 2}
```

The function `dict()` creates an empty dictionary.

Other useful dictionary methods include `values()`, `items()`, `has_key()`:

```
>>> d.values()
['19 deg C', '22 deg C']
>>> d.items()
[('yesterday', '19 deg C'), ('today', '22 deg C')]
>>> d.has_key('today')
True
>>> d.has_key('tomorrow')
```

```
False
>>> d.get('today','unknown')
'22 deg C'
>>> d.get('tomorrow','unknown')
'unknown'
>>> d.has_key('today')
True
>>> d.has_key('tomorrow')
False
>>> 'today' in d                  # as d.haskey('today')
True
>>> 'tomorrow' in d               # as d.haskey('tomorrow')
False
```

The method `get(key,default)` will provide the value for a given `key` if that key exists, otherwise it will return the `default` object.

Here is a more complex example:

```
order = {}          # create empty dictionary

#add orders as they come in
order['Peter'] = 'Pint of bitter'
order['Paul'] = 'Half pint of Hoegarden'
order['Mary'] = 'Gin Tonic'

#deliver order at bar
for person in order.keys():
    print person, "requests", order[person]
```

which produces this output:

```
Paul requests Half pint of Hoegarden
Peter requests Pint of bitter
Mary requests Gin Tonic
```

Some more technicalities:

- The keyword can be any (immutable) Python object. This includes:
  - ▷ numbers
  - ▷ strings
  - ▷ tuples.
- dictionaries are very fast in retrieving values (when given the key)

An other example to demonstrate an advantage of using dictionaries over pairs of lists:

```
dic = {}                          #create empty dictionary

dic["Hans"]    = "room 1033"      #fill dictionary
dic["Andy C"] = "room 1031"      #"Andy C" is key
dic["Ken"]     = "room 1027"      #"room 1027" is value

for key in dic.keys():
    print key, "works in", dic[key]
```

Output:

```
Hans works in room 1033
Andy C works in room 1031
Ken works in room 1027
```

Without dictionary:

```
people = ["Hans","Andy C","Ken"]
rooms  = ["room 1033","room 1031","room 1027"]

#possible inconsistency here since we have two lists
if not len( people ) == len( rooms ):
    raise RunTimeError,"people and rooms differ in length"

for i in range( len( rooms ) ):
    print people[i],"works in",rooms[i]
```

3.4 Passing arguments to functions

This section contains some more advanced ideas and makes use of concepts that are only later introduced in this text. The section may be more easily accessible at a later stage.

When objects are passed to a function, Python always passes (the value of) the reference to the object to the function. Effectively this is calling a function by reference, although one could refer to it as calling by value (of the reference).

We review argument passing by value and reference before discussing the situation in Python in more detail.

3.4.1 Call by value

One might expect that if we pass an object by value to a function, that modifications of that value inside the function will not affect the object (because we don't pass the object itself, but only its value, which is a copy). Here is an example of this behaviour (in C):

```
#include <stdio.h>

void pass_by_value(int m) {
  printf("in pass_by_value: received m=%d\n",m);
  m=42;
  printf("in pass_by_value: changed to m=%d\n",m);
}

int main(void) {
  int global_m = 1;
  printf("global_m=%d\n",global_m);
  pass_by_value(global_m);
  printf("global_m=%d\n",global_m);
  return 0;
}
```

together with the corresponding output:

```
global_m=1
in pass_by_value: received m=1
in pass_by_value: changed to m=42
global_m=1
```

The value 1 of the global variable `global_m` is not modified when the function `pass_by_value` changes its input argument to 42.

3.4.2 Call by reference

Calling a function by reference, on the other hand, means that the object given to a function is a reference to the object. This means that the function will see the same object as in the calling code (because they are referencing the same object: we can think of the reference as a pointer to the place in memory where the object is located). Any changes acting on the object inside the function, will then be visible in the object at the calling level (because the function does actually operate on the same object, not a copy of it).

Here is one example showing this using pointers in C:

```
#include <stdio.h>

void pass_by_reference(int *m) {
  printf("in pass_by_reference: received m=%d\n",*m);
  *m=42;
  printf("in pass_by_reference: changed to m=%d\n",*m);
}

int main(void) {
  int global_m = 1;
  printf("global_m=%d\n",global_m);
  pass_by_reference(&global_m);
  printf("global_m=%d\n",global_m);
  return 0;
}
```

together with the corresponding output:

```
global_m=1
in pass_by_reference: received m=1
in pass_by_reference: changed to m=42
global_m=42
```

C++ provides the ability to pass arguments as references by adding an ampersand in front of the argument name in the function definition:

```
#include <stdio.h>

void pass_by_reference(int &m) {
  printf("in pass_by_reference: received m=%d\n",m);
  m=42;
  printf("in pass_by_reference: changed to m=%d\n",m);
}

```

```
int main(void) {
  int global_m = 1;
  printf("global_m=%d\n",global_m);
  pass_by_reference(global_m);
  printf("global_m=%d\n",global_m);
  return 0;
}
```

together with the corresponding output:

```
global_m=1
in pass_by_reference: received m=1
in pass_by_reference: changed to m=42
global_m=42
```

3.4.3 Argument passing in Python

In Python, objects are passed as the value of a reference (think pointer) to the object. Depending on the way the reference is used in the function and depending on the type of object it references, this can result in pass-by-reference behaviour (where any changes to the object received as a function argument, are immediately reflected in the calling level).

Here are three examples to discuss this. We start by passing a list to a function which iterates through all elements in the sequence and doubles the value of each element:

```
def double_the_values(l):
    print "in double_the_values: l = %s" % l
    for i in range(len(l)):
        l[i] = l[i] * 2
    print "in double_the_values: changed l to l = %s" % l

l_global = [0, 1, 2, 3, 10]
print("In main: s=%s" % l_global)
double_the_values(l_global)
print("In main: s=%s" % l_global)
```

which produces this output:

```
In main: s=[0, 1, 2, 3, 10]
in double_the_values: l = [0, 1, 2, 3, 10]
in double_the_values: changed l to l = [0, 2, 4, 6, 20]
In main: s=[0, 2, 4, 6, 20]
```

The variable `l` is a reference to the list object. The line `l[i] = l[i] * 2` first evaluates the right-hand side and reads the element with index `i`, then multiplies this by two. A reference to this new object is then stored in the list object `l` at position with index `i`. We have thus modified the list object, that is referenced through `l`.

The reference to the list object does never change: the line

```
        l[i] = l[i] * 2
```

changes the elements `l[i]` of the list `l` but never changes the reference `l` for the list. Thus both the function and calling level are operating on the same object through the references `l` and `global_l`, respectively.

In contrast, here is an example where do not modify the elements of the list within the function:

```
def double_the_list(l):
    print "in double_the_list: l = %s" % l
    l = l + l
    print "in double_the_list: changed l to l = %s" % l

l_global = "Hello"
print("In main: l=%s" % l_global)
double_the_list(l_global)
print("In main: l=%s" % l_global)
```

which produces this output:

```
In main: l=Hello
in double_the_list: l = Hello
in double_the_list: changed l to l = HelloHello
In main: l=Hello
```

What happens here is that during the evaluation of `l = l + l` a new object is created that holds `l + l`, and that we then bind the name `l` to it. In the process, we lose the references to the list object `l` that was given to the function (and thus we do not change the list object given to the function).

Finally, let's look at

```
def double_the_value(l):
    print "in double_the_value: l = %s" % l
    l = 2 * l
    print "in double_the_values: changed l to l = %s" % l

l_global = 42
print("In main: s=%s" % l_global)
double_the_value(l_global)
print("In main: s=%s" % l_global)
```

which produces this output:

```
In main: s=42
in double_the_value: l = 42
in double_the_values: changed l to l = 84
In main: s=42
```

In this example, we also double the value (from 42 to 84) within the function. However, when we bind the object 84 to the python name `l` (that is the line `l = l * 2`) we have created a new object (84), and we bind the new object to `l`. In the process, we lose the reference to the object 42 within the function. This does not affect the object 42 itself, nor the reference `l_global` to it.

In summary, Python's behaviour of passing arguments to a function may appear to vary (if we view it from the pass by value versus pass by reference point of view). However, it is always call by value, where the value is a reference to the object in question, and the behaviour can be explained through the same reasoning in every case.

3.4.4 Performance considerations

Call by value function calls require copying of the value before it is passed to the function. From a performance point of view (both execution time and memory requirements), this can be an expensive

process if the value is large. (Imagine the value is a `numpy.array` object which could be several Megabytes or Gigabytes in size.)

One generally prefers call by reference for large data objects as in this case only a pointer to the data objects is passed, independent of the actual size of the object, and thus this is generally faster than call-by-value.

Python's approach of (effectively) calling by reference is thus efficient. However, we need to be careful that our function do not modify the data they have been given where this is undesired.

3.4.5 Inadvertent modification of data

Generally, a function should not modify the data given as input to it.

For example, the following code demonstrates the attempt to determine the maximum value of a list, and – inadvertently – modifies the list in the process:

```
def mymax(s):  # demonstrating side effect
    if len(s) == 0:
        raise ValueError('mymax() arg is an empty sequence')
    elif len(s) == 1:
        return s[0]
    else:
        for i in range(1, len(s)):
            if s[i] < s[i - 1]:
                s[i] = s[i - 1]
        return s[len(s) - 1]

s = [-45, 3, 6, 2, -1]
print("in main before caling mymax(s): s=%s" % s)
print("mymax(s)=%s" % mymax(s))
print("in main after calling mymax(s): s=%s" % s)
```

and produces this output

```
in main before caling mymax(s): s=[-45, 3, 6, 2, -1]
mymax(s)=6
in main after calling mymax(s): s=[-45, 3, 6, 6, 6]
```

The user of the `mymax()` function would not expect that the input argument is modified when the function executes. We should generally avoid this. There are several ways to find better solutions to the given problem:

- In this particular case, we could use the Python in-built function `max()` to obtain the maximum value of a sequence.
- If we felt we need to stick to storing temporary values inside the list [this is actually not necessary], we could create a copy of the incoming list `s` first, and then proceed with the algorithm (see section 3.4.6 on Copying objects).
- Use another algorithm which uses an extra temporary variable rather than abusing the list for this. For example:

```
def mymax(s):
    assert len(s) > 0, "mymax() arg is an empty sequence"
    tmp = s[0]
    for item in s:
```

```
        if item > tmp:
            tmp = item
    return tmp
```

- We could pass a tuple (instead of a list) to the function: a tuple is *immutable* and can thus never be modified (this would result in an exception being raised when the function tries to write to elements in the tuple).

3.4.6 Copying objects

Python provides the `id()` function which returns an integer number that is unique for each object. (In the current CPython implementation, this is the memory address.) We can use this to identify whether two objects are the same.

To copy a sequence object (including lists), we can slice it, *i.e.* if `a` is a list, then `a[:]` will return a copy of `a`. Here is a demonstration:

```
>>> a = range(10)
a
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = a
>>> b[0] = 42
>>> a                   # changing b changes a
[42, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> id(a)
4327533384
>>> id(b)
4327533384              # a and b refer to the same object
>>> c = a[:]
>>> id(c)               # c is a different object
4327533816
>>> c[0] = 100
>>> a                   # changing c does not affect a
[42, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python's standard library provides the `copy` module, which provides copy functions that can be used to create copies of objects. We could have used `import copy; c = copy.deepcopy(a)` instead of `c = a[:]`.

3.5 Equality and Identity/Sameness

A related question concerns the equality of objects.

3.5.1 Equality

The operators `<`, `>`, `==`, `>=`, `<=`, and `!=` compare the *values* of two objects. The objects need not have the same type. For example:

```
>>> a = 1.0; b = 1
>>> type(a)
<type 'float'>
>>> type(b)
```

```
<type 'int'>
>>> a == b
True
```

So the `==` operator checks whether the values of two objects are equal.

3.5.2 Identity / Sameness

To see check whether two objects `a` and `b` are the same (i.e. `a` and `b` are references to the same place in memory), we can use the `is` operator (continued from example above):

```
>>> a is b
False
```

Of course they are different here, as they are not of the same type.

We can also ask the `id` function which, according to the documentation string in Python 2.7 "*Returns the identity of an object. This is guaranteed to be unique among simultaneously existing objects. (Hint: it's the object's memory address.)*"

```
>>> id(a)
4298197712
>>> id(b)
4298187624
```

which shows that `a` and `b` are stored in different places in memory.

3.5.3 Example: Equality and identity

We close with an example involving lists:

```
>>> x = [0, 1, 2]
>>> y = x
>>> x == y
True
>>> x is y
True
>>> id(x)
4300880064
>>> id(y)
4300880064
>>>
```

Here, `x` and `y` are references to the same piece of memory, they are thus identical and the `is` operator confirms this. The important point to remember is that line 2 (`y=x`) creates a new reference `y` to the same list object that `x` is a reference for.

Accordingly, we can change elements of `x`, and `y` will change simultaneously as both `x` and `y` refer to the same object:

```
>>> x
[0, 1, 2]
>>> y
[0, 1, 2]
>>> x is y
True
```

```
>>> x[0] = 100
>>> y
[100, 1, 2]
>>> x
[100, 1, 2]
```

In contrast, if we use `z=x[:]` (instead of `z=x`) to create a new name `z`, then the slicing operation `x[:]` will actually create a copy of the list `x`, and the new reference `z` will point to the copy. The *value* of `x` and `z` is equal, but `x` and `z` are not the same object (they are not identical):

```
>>> x
[100, 1, 2]
>>> z = x[:]               # create copy of x before assigning to z
>>> z == x                # same value
True
>>> z is x                # are not the same object
False
>>> id(z)                 # confirm by looking at ids
4300737136
>>> id(x)
4300880064
>>> x
[100, 1, 2]
>>> z
[100, 1, 2]
```

Consequently, we can change `x` without changing `z`, for example (continued)

```
>>> x[0] = 42
>>> x
[42, 1, 2]
>>> z
[100, 1, 2]
```

Chapter 4

Introspection

A Python code can ask and answer questions about itself and the objects it is manipulating.

4.1 dir()

`dir()` is a built-in function which returns a list of all the names belonging to some namespace.

- If no arguments are passed to dir (i.e. `dir()`), it inspects the namespace in which it was called.
- If `dir` is given an argument (i.e. `dir(<object>)`, then it inspects the namespace of the object which it was passed.

For example:

```
>>> apples = ['Cox', 'Braeburn', 'Jazz']
>>> dir(apples)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__setslice__', '__str__', 'append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir()
['__builtins__', '__doc__', '__name__', 'apples']
>>> name = "Peter"
>>> dir(name)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__str__', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'replace', 'rfind',
'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
```

```
'zfill']
```

4.1.1 Magic names

You will find many names which both start and end with a double underscore (e.g. `__name__`). These are called magic names. Functions with magic names provide the implementation of particular python functionality.

For example, the application of the `str` to an object `a`, i.e. `str(a)`, will – internally – result in the method `a.__str__()` being called. This method `__str__` generally needs to return a string. The idea is that the `__str__()` method should be defined for all objects (including those that derive from new classes that a programmer may create) so that all objects (independent of their type or class) can be printed using the `str()` function. The actual conversion of some object `x` to the string is then done via the object specific method `x.__str__()`.

We can demonstrate this by creating a class `my_int` which inherits from the Python's integer base class, and overrides the `__str__` method. (It requires more Python knowledge than provided up to this point in the text to be able to understand this example.)

```
class my_int(int):
    """Inherited from int"""
    def __str__(self):
        """Tailored str representation of my int"""
        return "my_int: %s" % (int.__str__(self))

a = my_int(3)
b = int(4)             # equivalent to b = 4
print "a * b = ", a * b
print "Type a = ", type(a), "str(a) = ", str(a)
print "Type b = ", type(b), "str(b) = ", str(b)
```

This program produces the following output:

```
a * b = 12
Type a = <class '__main__.my_int'> str(Sa)= my_int: 3
Type b = <type 'int'> str(Sb)= 4
```

Further Reading

See →Python documentation, Data Model

4.2 type

The `type(<object>)` command returns the type of an object:

```
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
>>> type("Python")
<type 'str'>
>>> import math
>>> type(math)
```

```
<type 'module'>
>>> type(math.sin)
<type 'builtin_function_or_method'>
```

4.3 isinstance

`isinstance(<object>, <typespec>)` returns True if the given object is an instance of the given type, or any of its superclasses. Use `help(isinstance)` for the full syntax.

```
>>> isinstance(2,int)
True
>>> isinstance(2.,int)
False
>>> isinstance(a,int)     # a is an instance of my_int
                          # which inherits from int
True
>>> type(a)
<class '__main__.my_int'>
```

4.4 help

- The `help(<object>)` function will report the docstring (magic attritube with name `__doc__`) of the object that it is given, sometimes complemented with additional information. In the case of functions, `help` will also show the list of arguments that the function accepts (but it cannot provide the return value).
- `help()` starts an interactive help environment.
- It is common to use the `help` command a lot to remind oneself of the syntax and semantic of commands.

```
>>> help(isinstance)
Help on built-in function isinstance in module __builtin__:

isinstance(...)
    isinstance(object, class-or-type-or-tuple) -> bool

    Return whether an object is an instance of a class or of a
    subclass thereof. With a type as second argument, return
    whether that is the object's type. The form using a tuple,
    isinstance(x, (A, B, ...)), is a shortcut for
    isinstance(x, A) or isinstance(x, B) or ... (etc.).

>>> import math
>>> help(math.sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)
```

```
    Return the sine of x (measured in radians).

>>> help(math)
Help on module math:

NAME
    math

FILE
    /sw/lib/python2.4/lib-dynload/math.so

MODULE DOCS
    http://www.python.org/doc/current/lib/module-math.html

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

[... snip: content removed here ...]

    tanh(...)
        tanh(x)

        Return the hyperbolic tangent of x.

DATA
    e = 2.7182818284590451
    pi = 3.1415926535897931
```

The help function needs to be given the name of an object (which must exist in the current name space). For example pyhelp(math.sqrt) will not work if the math module has not been imported before

```
>>> help(math.sqrt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> import math
>>> help(math.sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Instead of importing the module, we could also have given the *string* of `math.sqrt` to the help function, i.e.:

```
>>> help('math.sqrt')
Help on built-in function sqrt in module math:
...
```

`help` is a function which gives information about the object which is passed as its argument. Most things in Python (classes, functions, modules, etc.) are objects, and therefor can be passed to help. There are, however, some things on which you might like to ask for help, which are not existing Python objects. In such cases it is often possible to pass a string containing the name of the thing or concept to help, for example

- `help('modules')` will generate a list of all modules which can be imported into the current interpreter. Note that help(modules) (note absence of quotes) will result in a NameError (unless you are unlucky enough to have a variable called modules floating around, in which case you will get help on whatever that variable happens to refer to.)
- `help('some_module')`, where some_module is a module which has not been imported yet (and therefor isn't an object yet), will give you that module's help information.
- `help('some_keyword')`: For example `and`, `if` or `print` (*i.e.* `help('and')`, `help('if')` and `help('print')`). These are special words recognized by Python: they are not objects and thus cannot be passed as arguments to help. Passing the name of the keyword as a string to help works, but only if you have Python's HTML documentation installed, and the interpreter has been made aware of its location by setting the environment variable PYTHONDOCS.

4.5 Docstrings

The command `help(<object>)` accesses the documentation strings of objects.

Any literal string apparing as the first item in the definition of a class, function, method or module, is taken to be its *docstring*.

`help` includes the docstring in the information it displays about the object.

In addition to the docstring it may display some other information, for example, in the case of functions, it displays the function's signature.

The docstring is stored in the object's `__doc__` attribute.

```
>>> help(math.sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

```
>>> print math.sin.__doc__
sin(x)

Return the sine of x (measured in radians).
```

For user-defined functions, classes, types, modules, ...), one should always provide a docstring. Documenting a user-provided function:

```
>>> def power2and3(x):
...     """Returns the tuple (x**2, x**3)"""
...     return x**2 ,x**3
...
>>> power2and3(2)
(4, 8)
>>> power2and3(4.5)
(20.25, 91.125)
>>> power2and3(0+1j)
((-1+0j), -1j)
>>> help(power2and3)
Help on function power2and3 in module __main__:

power2and3(x)
    Returns the tuple (x**2, x**3)

>>> print power2and3.__doc__
Returns the tuple (x**2,x**3)
```

Chapter 5

Input and Output

In this section, we describe the old style (pre Python 3) of printing, which includes the use of the `print` command without parenthesis (which is not allowed to be used in Python 3.x), and the old-style format specifiers `%` (which can be used in Python 2.x and 3.x). See section 5.1.5 for details.

5.1 Printing to standard output (normally the screen)

The `print` command is the most commonly used command to print information to the "standard output devices" which is normally the screen.

There are two modes to use print.

5.1.1 Simple print (not compatible with Python 3.x)

The easiest way to use the print command is to list the variables to be printed, separated by comma. Here are a few examples:

```
>>> a = 10
>>> b = 'test text'
>>> print a
10
>>> print b
test text
>>> print a, b
10 test text
>>> print "The answer is",a
The answer is 10
>>> print "The answer is",a,"and the string contains",b
The answer is 10 and the string contains test text
>>> print "The answer is",a,"and the string reads",b
The answer is 10 and the string reads test text
```

Python adds a space between every object that is being printed.

Python prints a new line after every print statement. To suppress that, add an extra comma at the end of the line:

```
print "Printing in line one",
print "...still printing in line one."
```

5.1.2 Formatted printing

The more sophisticated way of formatting output uses a syntax very similar to Matlab's `fprintf` (and therefor also similar to C's `printf`).

The overall structure is that there is a string containing format specifiers, followed by a percentage sign and a tuple that contains the variables to be printed in place of the format specifiers.

```
>>> print "a = %d b = %d" % (10,20)
a = 10 b = 20
```

A string can contain format identifiers (such as `%f` to format as a floats, `%d` to format as an integers, and `%s` to format as a string):

```
>>> from math import pi
>>> print "Pi = %5.2f" % pi
Pi =  3.14
>>> print "Pi = %10.3f" % pi
Pi =      3.142
>>> print "Pi = %10.8f" % pi
Pi = 3.14159265
>>> print "Pi = %d" % pi
Pi = 3
```

The format specifier of type `%W.Df` means that a Float should be printed with a total Width of `W` characters and `D` digits behind the Decimal point. (This is identical to Matlab and C, for example.)

To print more than one object, provide multiple format specifiers and list several objects in the tuple:

```
>>> print "Pi = %f, 142*pi = %f and pi^2 = %f." % (pi,142*pi,pi**2)
Pi = 3.141593, 142*pi = 446.106157 and pi^2 = 9.869604.
```

Note that the conversion of a format specifier and a tuple of variables into string does not rely on the `print` command:

```
>>> from math import pi
>>> "pi = %f" % pi
'pi = 3.141593'
```

This means that we can convert objects into strings whereever we need, and we can decide to print the strings later – there is no need to couple the formatting closely to the code that does the printing.

Overview of commonly used format specifiers using the astronomical unit as an example:

```
>>> AU = 149597870700  # astronomical unit [m]
>>> "%f" % AU          # line 1 in table
'149597870700.000000'
```

specifier	style	Example output for AU
`%f`	floating point	149597870700.000000
`%e`	exponential notation	1.495979e+11
`%g`	shorter of %e or %f	1.49598e+11
`%d`	integer	149597870700
`%s`	`str()`	149597870700
`%r`	`repr()`	149597870700L

5.1.3 "str" and "__str__"

All objects in Python should provide a method `__str__` which returns a nice string representation of the object. This method `a.__str__()` is called when we apply the `str` function to object `a`:

```
>>> a = 3.14
>>> a.__str__()
'3.14'
>>> str(a)
'3.14'
```

The `str` function is extremely convenient as it allows us to print more complicated objects, such as

```
>>> b = [3, 4.2, ['apple', 'banana'], (0, 1)]
>>> str(b)
"[3, 4.2, ['apple', 'banana'], (0, 1)]"
```

The way Python prints this is that it uses the `__str__` method of the list object. This will print the opening square bracket `[` and then call the `__str__` method of the first object, i.e. the integer 3. This will produce `3`. Then the list object's `__str__` method prints the comma `,` and moves on to call the `__str__` method of the next element in the list (i.e. `4.2`) to print itself. This way any composite object can be represented as a string by asking the objects it holds to convert themselves to strings.

The string method of object `x` is called implicitly, when we

- use the "%s" format specifier to print `x`
- pass the object `x` directly to the print command:

```
>>> print(b)
[3, 4.2, ['apple', 'banana'], (0, 1)]
>>> print("%s" % b)
[3, 4.2, ['apple', 'banana'], (0, 1)]
```

5.1.4 "repr" and "__repr__"

A second function, `repr`, should convert a given object into a string presentation *so that this string can be used to re-created the object using the* `eval` *function*. The `repr` function will generally provide a more detailed string than `str`. Applying `repr` to the object `x` will attempt to call `x.__repr__()`.

```
>>> from math import pi as a1
>>> str(a1)
'3.14159265359'                # convenient presentation as string
>>> repr(a1)
'3.141592653589793'            # exact representation as string
>>> number_as_string = repr(a1)
>>> a2 = eval(number_as_string)  # evaluate string
>>> a2
3.141592653589793
>>> a2-a1                      # -> repr is exact representation
0.0
>>> a1-eval(repr(a1))
0.0
>>> a1-eval(str(a1))           # -> str has lost a few digits
-2.0694557179012918e-13
```

We can convert an object to its `str()` or `repr` presentation using the format specifiers `%s` and `%r`, respectively.

```
>>> import math
>>> "%s" % math.pi
'3.14159265359'
>>> "%r" % math.pi
'3.141592653589793'
```

5.1.5 Changes from Python 2 to Python 3: `print`

One (maybe the most obvious) change going from Python 2 to Python 3 is that the `print` command loses its special status. In Python 2, we could print "Hello World" using

```
print "Hello World"                    # valid in Python 2.x
```

Effectively, we call the function `print` with the argument `"Hello World"`. All other functions in Python are called such that the argument is enclosed in parentheses, i.e.

```
print("Hello World")                   # valid in Python 3.x
```

This is the new convention *required* in Python 3 (and *allowed* for recent version of Python 2.x.)

The good news is that everything we have learned about formatting strings using the percentage operator, can be used in Python 3.x as well:

```
>>> import math
>>> a = math.pi
>>> "my pi = %f" % a           # string formatting
'my pi = 3.141593'
>>> print "my pi = %f" % a     # valid print in 2.x
my pi = 3.141593
>>> print("my pi = %f" % a)    # valid print in 2.7 and 3.x
my pi = 3.141593

# another example to follow
>>> "Short pi = %.2f, longer pi = %.12f." % (a, a)
'Short pi = 3.14, longer pi = 3.141592653590.'
>>> print "Short pi = %.2f, longer pi = %.12f." % (a, a)
Short pi = 3.14, longer pi = 3.141592653590.
>>> print("Short pi = %.2f, longer pi = %.12f." % (a, a))
Short pi = 3.14, longer pi = 3.141592653590.
```

5.1.6 Changes from Python 2 to Python 3: formatting of strings

A new system of built-in formatting has been proposed and is meant to replace the old-style percentage operator formatting (%) in the long term.

Basic ideas in examples:

```
>>> "{} needs {} pints".format('Peter',4)       # insert values in order
'Peter needs 4 pints'
>>> "{0} needs {1} pints".format('Peter',4)     # index which element
'Peter needs 4 pints'                           # from tuple to insert
```

```
>>> "{1} needs {0} pints".format('Peter',4)
'4 needs Peter pints'
>>> "{name} needs {number} pints".format(\     # reference element to
...                 name='Peter',number=4)     # print by name
'Peter needs 4 pints'
>>> "Pi is approximately {:f}.".format(math.pi)      # can use old-
'Pi is approximately 3.141593.'                      # style format %f
>>> "Pi is approximately {:.2f}.".format(math.pi)    # and precision
'Pi is approximately 3.14.'
>>> "Pi is approximately {:6.2f}.".format(math.pi)   # and width
'Pi is approximately   3.14.'                        # specification
```

This is a powerful and elegant way of string formatting, which is meant to be used exclusively in the far future.

Further information

- Examples http://docs.python.org/library/string.html#format-examples
- Python Enhancement Proposal 3101
- Python library String Formatting Operations
- Old string formatting
- Introduction to Fancier Output Formatting, Python tutorial, section 7.1

5.2 Reading and writing files

Here is a program that

1. writes some text to a file with name `test.txt`,
2. and then reads the text again and
3. prints it to the screen.

```
# 1. Write a file
out_file = open("test.txt", "w")           #'w' stands for Writing
out_file.write("Writing text to file. This is the first line.\n"+\
               "And the second line.")
out_file.close()                           #close the file

# 2. Read a file
in_file = open("test.txt", "r")            #'r' stands for Reading
text = in_file.read()                      #read complete file into
                                           #string variable text
in_file.close()                            #close the file

# 3. Display data
print text
```

The data stored in the file `test.txt` is

```
Writing text to file. This is the first line.
And the second line.
```

In more detail, you have opened a file with the `open` command, and assigned this open file object to the variable `out_file`. We have then written data to the file using the `out_file.write` method. Note that in the example above, we have given a string to the `write` method. We can, of course, use all the formatting that we have discussed before in section 5.1.2. For example, to write this file with name table `table.txt`

```
 1 x 17 =   17
 2 x 17 =   34
 3 x 17 =   51
 4 x 17 =   68
 5 x 17 =   85
 6 x 17 =  102
 7 x 17 =  119
 8 x 17 =  136
 9 x 17 =  153
10 x 17 =  170
```

we can use this Python program

```
f = open('table.txt', 'w')
for i in range(1, 11):
    f.write("%2d x 17 = %4d\n" % (i, i * 17))
f.close()
```

It is good practice to `close()` files when we have finished reading and writing. If a Python program is left in a controlled way (i.e. not through a power cut or an unlikely bug deep in the Python language or the operating system) then it will close all open files as soon as the file objects are destroyed. However, closing them actively as soon as possible is better style.

5.2.1 File reading examples

We use a file named `myfile.txt` containing the following 3 lines of text for the examples below:

```
This is the first line.
This is the second line.
This is a third and last line.
```

fileobject.read()

The `fileobject.read()` method reads the whole file, and returns it as one string (including new line characters).

```
>>> f = open('myfile.txt', 'r')
>>> f.read()
'This is the first line.\nThis is the second line.\nThis is a third and last line.\n'
>>> f.close()
```

fileobject.readlines()

The `fileobject.readlines()` method returns a list of strings, where each element of the list corresponds to one line in the string:

```
>>> f = open('myfile.txt', 'r')
>>> f.readlines()
['This is the first line.\n', 'This is the second line.\n',
 'This is a third and last line.\n']
>>> f.close()
```

This is often used to iterate over the lines, and to do something with each line. For example:

```
f = open('myfile.txt', 'r')
for line in f.readlines():
    print("%d characters" % len(line))
f.close()
```

with output

```
24 characters
25 characters
31 characters
```

Note that this will read the complete file into a list of strings when the `readlines()` method is called. This is no problem if we know that the file is small and will fit into the machine's memory.

If so, we can also close the file before we process the data, i.e.:

```
f = open('myfile.txt', 'r')
lines = f.readlines():
f.close()
for line in lines:
    print("%d characters" % len(line))
```

Iterating over lines (file object)

There is a neater possibility to read a file line by line which (i) will only read one line at a time (and is thus suitable for large files as well) and (ii) results in more compact code:

```
f = open('myfile.txt', 'r')
for line in f:
    print("%d characters" % len(line))
f.close()
```

Here, the file handler `f` acts as in iterator and will return the next line in every subsequent iteration of the for-loop until the end of the file is reached (and then the for-loop is terminated).

Further reading Methods of File objects, Tutorial, Section 7.2.1

Chapter 6

Control Flow

6.1 Basics

For a given file with a python program, the python interpreter will start at the top and then process the file. We demonstrate this with a simple program, for example:

```
1 def f(x):
2     """function that computes and returns x*x"""
3     return x * x
4
5 print("Main program starts here")
6 print("4 * 4 = %s" % f(4))
7 print("In last line of program -- bye")
```

The basic rule is that commands in a file (or function or any sequence of commands) is processed from top to bottom. If several commands are given in the same line (separated by `;`), then these are processed from left to right (although it is discouraged to have multiple statements per line to maintain good readability of the code.)

In this example, the interpreter starts at the top (line 1). It finds the `def` keyword and remembers for the future that the function `f` is defined here. (It will not yet execute the function body, i.e. line 3 – this only happens when we call the function.) The interpreter can see from the indentation where the body of the function stops: the indentation in line 5 is different from that of the first line in the function body (line2), and thus the function body has ended, and execution should carry on with that line. (Empty lines do not matter for this analysis.)

In line 5 the interpreter will print the output `Main program starts here`. Then line 6 is executed. This contains the expression `f(4)` which will call the function `f(x)` which is defined in line 1 where `x` will take the value `4`. [Actually `x` is a reference to the object `4`.] The function `f` is then executed and computes and returns `4*4` in line 3. This value `16` is used in line 6 to replace `f(4)` and then the string representation `%s` of the object 16 is printed as part of the print command in line 6.

The interpreter then moves on to line 7 before the program ends.

We will now learn about different possibilities to direct this control flow further.

6.1.1 Conditionals

The python values `True` and `False` are special inbuilt objects:

```
>>> a = True
>>> print a
True
>>> type(a)
```

```
<type 'bool'>
>>> b = False
>>> print b
False
>>> type(b)
<type 'bool'>
```

We can operate with these two logical values using boolean logic, for example the logical and operation (`and`):

```
>>> True and True          #logical and operation
True
>>> True and False
False
>>> False and True
False
>>> True and True
True
>>> c = a and b
>>> print c
False
```

There is also logical or (`or`) and the negation (`not`):

```
>>> True or False
True
>>> not True
False
>>> not False
True
>>> True and not False
True
```

In computer code, we often need to evaluate some expression that is either true or false (sometimes called a "predicate"). For example:

```
>>> x = 30          # assign 30 to x
>>> x > 15          # is x greater than 15
True
>>> x > 42
False
>>> x == 30         # is x the same as 30?
True
>>> x == 42
False
>>> not x == 42     # is x not the same as 42?
True
>>> x != 42         # is x not the same as 42?
True
>>> x > 30          # is x greater than 30?
False
>>> x >= 30   # is x greater than or equal to 30?
True
```

6.2 If-then-else

Further information

- Introduction to If-then in Python tutorial, section 4.1

The `if` statement allows conditional execution of code, for example:

```
a = 34
if a > 0:
    print "a is positive"
```

The if-statement can also have an `else` branch which is executed if the condition is wrong:

```
a = 34
if a > 0:
    print "a is positive"
else:
    print "a is non-positive (i.e. negative or zero)"
```

Finally, there is the `elif` (read as "else if") keyword that allows checking for several (exclusive) possibilities:

```
a = 17
if a == 0:
    print "a is zero"
elif a < 0:
    print "a is negative"
else:
    print "a is positive"
```

6.3 For loop

Further information

- Introduction to for-loops in Python tutorial, section 4.2

The `for`-loop allows to iterate over a sequence (this could be a string or a list, for example). Here is an example:

```
>>> for animal in ['dog','cat','mouse']:
...     print animal, animal.upper()
dog DOG
cat CAT
mouse MOUSE
```

Together with the `range()` command (3.3.2), one can iterate over increasing integers:

```
>>> for i in range(5,10):
...     print i
5
6
7
8
9
```

6.4 While loop

The `while` keyword allows to repeat an operation while a condition is true. Suppose we'd like to know for how many years we have to keep 100 pounds on a savings account to reach 200 pounds simply due to annual payment of interest at a rate of 5%. Here is a program to compute that this will take 15 years:

```
mymoney = 100          # in GBP
rate = 1.05            # 5% interest
years = 0
while mymoney < 200:   # repeat until 20 pounds reached
    mymoney = mymoney * rate
    years = years + 1
print 'We need', years, 'years to reach', mymoney, 'pounds.'
```

produces

```
We need 15 years to reach 207.892817941 pounds.
```

6.5 Relational operators (comparisons) in `if` and `while` statements

The general form of `if` statements and `while` loops is the same: following the keyword `if` or `while`, there is a *condition* followed by a colon. In the next line, a new (and thus indented!) block of commands starts that is executed if the condition is True).

For example, the condition could be equality of two variables `a1` and `a2` which is expressed as `a1==a2`:

```
a1 = 42
a2 = 42
if a1 == a2:
    print("a1 and a2 are the same")
```

Another example is to test whether `a1` and `a2` are not the same. For this, we have two possibilities. Option number 1 uses the *inequality operator* `!=`:

```
if a1 != a2:
    print("a1 and a2 are different")
```

Option two uses the keyword `not` in front of the condition:

```
if not a1 == a2:
    print("a1 and a2 are different")
```

Comparisons for "greater" (`>`), "smaller" (`<`) and "greater equal" (`>=`) and "smaller equal" (`<=`) are straightforward.

Finally, we can use the logical operators "`and`" and "`or`" to combine conditions:

```
if a > 10 and b > 20:
    print "A is greater than 10 and b is greater than 20"
if a > 10 or b < -5:
    print "Either a is greater than 10, or "\
          "b is smaller than -5, or both."
```

Use the Python prompt to experiment with these comparisons and logical expressions. For example:

```
>>> T = -12.5
>>> if T < -20:
...     print "very cold"
...
>>> if T < -10:
...     print "quite cold"
...
quite cold
>>> T < -20
False
>>> T < -10
True
>>>
```

6.6 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not necessarily fatal: exceptions can be *caught* and dealt with within the program. Most exceptions are not handled by programs, however, and result in error messages as shown here

```
  >>> 10 * (1/0)
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  ZeroDivisionError: integer division or modulo by zero
  >>> 4 + spam*3
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  NameError: name 'spam' is not defined
  >>> '2' + 2
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  TypeError: cannot concatenate 'str' and 'int' objects
```

Schematic exception catching with all options

```
try:
   # code body
except ArithmeticError:
   # what to do if arithmetic error
except IndexError, the_exception:
   # the_exception refers to the exeption in this block
except:
   # what to do for ANY other exception
else:  # optional
   # what to do if no exception raised
try:
   # code body
```

```
finally:
    # what to do ALWAYS
```

Starting with Python 2.5, you can use the with statement to simplify the writing of code for some predefined functions, in particular the `open` function to open files: see http://docs.python.org/tutorial/errors.html#predefined-clean-up-actions.

Example: We try to open a file that does not exist, and Python will raise an exception of type `IOError` which stands for Input Output Error:

```
>>> f = open("filenamethatdoesnotexist", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'filenamethatdoesnotexist'
```

If we were writing an application with a userinterface where the user has to type or select a filename, we would not want to application to stop if the file does not exist. Instead, we need to catch this exception and act accordingly (for example by informing the user that a file with this filename does not exist and ask whether they want to try another file name). Here is the skeleton for catching this exception:

```
>>> try:
...     f = open("filenamethatdoesnotexist","r")
... except IOError:
...     print "Could not open that file"
...
Could not open that file
```

There is a lot more to be said about exceptions and their use in larger programs. Start reading Python Tutorial Chapter 8: Errors and Exceptions if you are interested.

6.6.1 Raising Exceptions

Raising exception is also referred to as 'throwing an exception'.

Possibilities of raising an Exception

- `raise OverflowError`
- `raise OverflowError, "Bath is full"` (Old style, now discouraged)
- `raise OverflowError("Bath is full")`
- `e = OverflowError("Bath is full"); raise e`

Exception hierarchy

The standard exceptions are organized in an inheritance hierarchy e.g. OverflowError is a subclass of ArithmeticError (not BathroomError); this can be seen when looking at `help('exceptions')` for example.

You can derive your own exceptions from any of the standard ones. It is good style to have each module define its own base exception.

6.6.2 Creating our own exceptions

- You can and should derive your own exceptions from the built-in Exception.
- To see what built-in exceptions exist, look in the module exceptions (try `help('exceptions')`), or go to http://docs.python.org/library/exceptions.html#bltin-exceptions.

6.6.3 LBYL vs EAFP

- LBYL (Look Before You Leap) vs
- EAFP (Easer to ask forgiveness than permission)

Example for LBYL:

```
if denominator == 0:
    print "Oops"
else:
    print numerator/denominator
```

Easier to Ask for Forgiveness than Permission:

```
try:
    print numerator/denominator
except ZeroDivisionError:
    print "Oops"
```

The Python documentation says about EAFP:

> Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

Source: http://docs.python.org/glossary.html#term-eafp
The Python documentation says about LBYL:

> Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.
>
> In a multi-threaded environment, the LBYL approach can risk introducing a race condition between the looking and the leaping. For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

Source: http://docs.python.org/glossary.html#term-lbyl
EAFP is the Pythonic way.

Chapter 7

Functions and modules

7.1 Introduction

Functions allow us to group a number of statements into a logical block. We communicate with a function through a clearly defined interface, providing certain parameters to the function, and receiving some information back. Apart from this interface, we generally do not how exactly a function does the work to obtain the value it returns

For example the function `math.sqrt`: we do not know how exactly it computes the square root, but we know about the interface: if we pass x into the function, it will return (an approximation of) $\sqrt{x}$.

This abstraction is a useful thing: it is a common technique in engineering to break down a system into smaller (black-box) components that all work together through well defined interfaces, but which do not need to know about the internal realisations of each other's functionality. In fact, not having to care about these implementation details can help to have a clearer view of the system composed of many of these components.

Functions provide the basic building blocks of functionality in larger programs (and computer simulations), and help to control the inherent complexity of the process.

We can group functions together into a Python module (see section 7.5), and in this way create our own libraries of functionality.

7.2 Using functions

The word "function" has different meanings in mathematics and programming. In programming it refers to a named sequence of operations that perform a computation. For example, the function `sqrt()` which is defined in the `math` module computes the square root of a given value:

```
>>> from math import sqrt
>>> sqrt(4)
2.0
```

The value we pass to the function `sqrt` is 4 in this example. This value is called the *argument* of the function. A function may have more than one argument.

The function returns the value 2.0 (the result of its computation) to the "calling context". This value is called the *return value* of the function.

It is common to say that a function *takes* an argument and *returns* a result or return value.

Common confusion about printing and returning values

It is a common beginner's mistake to confuse the *printing* of values with *returning* values. In the following example it is impossible to see whether the function `math.sin` returns a value or whether it prints the value:

```
>>> import math
>>> math.sin(2)
0.9092974268256817l
>>>
```

We import the `math` module, and call the `math.sin` function with an argument of `2`. The `math.sin(2)` call will actually *return* the value `0.909...` not print it. However, because we have not assigned the return value to a variable, the Python prompt will print the returned object.

The following alternative sequence works only if the value is returned:

```
>>> x = math.sin(2)
>>> print x
0.909297426826
```

The return value of the function call `math.sin(2)` is assigned to the variable `x`, and `x` is printed in the next line.

Generally, functions should execute "silently" (i.e. not print anything) and report the result of their computation through the return value.

Part of the confusion about printed versus return values at the Python prompt comes from the Python prompt printing (a representation) of returned objects *if* the returned objects are not assigned. Generally, seeing the returned objects is exactly what we want (as we normally care about the returned object), just when learning Python this may cause mild confusion about functions returning values or printing values.

Further information

- Think Python has a gentle introduction to functions (on which the previous paragraph is based) in chapter 3 (Functions) and chapter 6 (Fruitful functions).

7.3 Defining functions

The generic format of a function definitions:

```
def my_function(arg1, arg2, ..., argn):
    """Optional docstring."""

    # Implementation of the function

    return result   # optional

#this is not part of the function
some_command
```

Allen Downey's terminology (in his book Think Python) of fruitful and fruitless functions distinguishes between functions that return a value, and those that do not return a value. The distinction refers to whether a function provides a return value (=fruitful) or whether the function does not explicitly return a value (=fruitless). If a functions does not make use of the `return` statement, we

tend to say that the function returns nothing (whereas in reality in will always return the `None` object when it terminates – even if the `return` statement is missing).

For example, the function `greeting` will print "Hello World" when called (and is fruitless as it does not return a value).

```
def greeting():
    print "Hello World!"
```

If we call that function:

```
>>> greeting()
Hello World!
```

it prints "Hello World" to stdout, as we would expect. If we assign the return value of the function to a variable `x`, we can inspect it subsequently:

```
>>> x = greeting()
Hello World!
>>> print x
None
```

and find that the `greeting` function has indeed returned the `None` object.

Another example for a function that does not return any value (that means there is no `return` keyword in the function) would be:

```
def printpluses(n):
   print n * "+"
```

Generally, functions that return values are more useful as these can be used to assemble code (maybe as another function) by combining them cleverly. Let's look at some examples of functions that do return a value.

Suppose we need to define a function that computes the square of a given variable. The function source could be:

```
def square(x):
    return x * x
```

The keyword `def` tells Python that we are *defining* a function at that point. The function takes one argument (`x`). The function returns `x*x` which is of course x^2. Here is the listing of a file that shows how the function can be defined and used: (note that the numbers on the left are line numbers and are not part of the program)

```
1 def square(x):
2     return x * x
3
4 for i in range(5):
5     i_squared = square(i)
6     print i, '*', i, '=', i_squared
```

It is worth mentioning that lines 1 and 2 define the square function whereas lines 4 to 6 are the main program.

This program will produce the following output

```
0 * 0 = 0
1 * 1 = 1
```

```
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
```

We can define functions that take more than one argument:

```
import math

def hypot(x, y):
    return math.sqrt(x * x + y * y)
```

It is also possible to return more than one argument. Here is an example of a function that converts a given string into all characters uppercase and all characters lowercase and returns the two versions. We have included the main program to show how this function can be called:

```
def upperAndLower(string):
    return string.upper(), string.lower()

testword = 'Banana'

uppercase, lowercase = upperAndLower(testword)

print testword, 'in lowercase:', lowercase,\
      'and in uppercase',uppercase
```

We can define multiple Python functions in one file. Here is an example with two functions:

```
def returnstars( n ):
    return n * '*'

def print_centred_in_stars( string ):
    linelength = 46
    starstring = returnstars((linelength - len(string)) / 2)

    print starstring + string + starstring

print_centred_in_stars('Hello world!')
```

This program produces

```
*****************Hello world!*****************
```

Further reading

- Python Tutorial: Section 4.6 Defining Functions

7.4 Default values and optional parameters

Python allows to define *default* values for function parameters. Here is an example:

```
def print_mult_table(n, upto=10):
    for i in range(1, upto + 1):
        print "%3d * %d = %4d" % (i, n, i * n)
```

```
print_mult_table(5)
print_mult_table(9, 3)
```

This program will print the following output when executed:

```
 1 * 5 =   5
 2 * 5 =  10
 3 * 5 =  15
 4 * 5 =  20
 5 * 5 =  25
 6 * 5 =  30
 7 * 5 =  35
 8 * 5 =  40
 9 * 5 =  45
10 * 5 =  50
 1 * 9 =   9
 2 * 9 =  18
 3 * 9 =  27
```

So how does it work? The function `print_mult_table` takes two arguments: `n` and `upto`. The first argument `n` is a "normal" variable. The second argument `upto` has a default value of 10. In other words: should the user of this function only provide one argument, then this provides the value for `n` and `upto` will default to 10. If two arguments are provided, the first one will be for `n` and the second for `upto` (as shown in the code example above).

7.5 Modules

Modules

- Group together functionality
- Provide namespaces
- Python's standard library contains a vast collection of modules - "Batteries Included"
  - ▷ Try `help('modules')`
- Means of extending Python

7.5.1 Importing modules

```
import math
```

This will introduce the name `math` into the namespace in which the import command was issued. The names within the `math` module will not appear in the enclosing namespace: they must be accessed through the name `math`. For example: `math.sin`.

```
import math, cmath
```

More than one module can be imported in the same statement, although the Python Style Guide recommends not to do this. Instead, we should write

```
import math
import cmath
```

```
import math as mathematics
```

The name by which the module is known locally can be different from its ”official” name. Typical uses of this are

- To avoid name clashes with existing names
- To change the name to something more manageable. For example `import SimpleHTTPServer as shs`. This is discouraged for production code (as longer meaningful names make programs far more understandable than short cryptic ones), but for interactively testing out ideas, being able to use a short synonym can make your life much easier. Given that (imported) modules are first class objects, you can, of course, simply do `shs = SimpleHTTPServer` in order to obtain the more easily typable handle on the module.

```
from math import sin
```

This will import the `sin` function from the `math` module, but it will not introduce the name math into the current namespace. It will only introduce the name `sin` into the current namespace. It is possible to pull in more than one name from the module in one go:

```
from math import sin, cos
```

Finally, let’s look at this notation:

```
from math import *
```

Once again, this does not introduce the name math into the current namespace. It does however introduce *all public names* of the math module into the current namespace. Broadly speaking, it is a bad idea to do this:

- Lots of new names will be dumped into the current namespace.
- Are you sure they will not clobber any names already present?
- It will be very difficult to trace where these names came from
- Having said that, some modules (including ones in the standard library, recommend that they be imported in this way). Use with caution!
- This is fine for interactive quick and dirty testing or small calculations.

7.5.2 Creating modules

A module is in principle nothing else than a python file. Here is an example of a module file which is saved in `module1.py`:

```
def someusefulfunction():
    pass

print "My name is", __name__
```

We can execute this (module) file as a normal python program (for example `python module1.py`), and the output is

```
My name is __main__
```

We note that the Python magic variable `__name__` takes the value `__main__` if the program file `module1.py` is executed.

On the other hand, we can *import* `module1.py` in another file (which could have the name `prog.py`), for example like this:

```
import module1                  #in file prog.py
```

When Python comes across the `import module1` statement in `prog.py`, it looks for the file `module1.py` in the current working directory (and if it can't find it there in all the directories in `sys.path`) and opens the file `module1.py`. While parsing the file `module1.py` from top to bottom, it will add any function definitions in this file into the `module1` name space in the calling context (that is the main program in `prog.py`). It this example, there is only the function `someusefulfunction`. Once the import process is completed, we can make use of `module1.someusefulfunction` in `prog.py`. If Python comes across statements other than function (and class) definitions while importing `module1.py`, it carries those out immediately. In this case, it will thus come across the statement `print "My name is",__name__`. The output of this reads now:

```
My name is module1
```

Note the difference to the output if we *import* `module1.py` rather than executing it on its own: `__name__` inside a module takes the value of the module name if the file is imported.

7.5.3 Use of __name__

In summary,

- `__name__` is `"__main__"` if the module file is run on its own
- `__name__` is the name of the module (i.e. the module filename without the `.py` suffix) if the module file is imported.

We can therefor use the following `if` statement in `module1.py` to write code that is *only run* when the module is executed on its own:

```
def someusefulfunction():
    pass

if __name__ == "__main__":
    print("I am running on my own")
```

This is useful to keep test programs or demonstrations of the abilities of a module in this "conditional" main program. It is common practice for any module files to have such a conditional main program which demonstrates its capabilities.

7.5.4 Example 1

The next example shows a main program for the another file `vectools.py` that is used to demonstrate the capabilities of the functions defined in that file:

```
from __future__ import division
import math

import numpy as N


def norm(x):
    """returns the magnitude of a vector x"""
    return math.sqrt(sum(x ** 2))


def unitvector(x):
    """returns a unit vector x/|x|. x needs to be a numpy array."""
    xnorm = norm(x)
    if xnorm == 0:
        raise ValueError("Can't normalise vector with length 0")
    return x / norm(x)


if __name__ == "__main__":
    #a little demo of how the functions in this module can be used:
    x1 = N.array([0, 1, 2])
    print("The norm of " + str(x1) + " is " + str(norm(x1)) + ".")
    print("The unitvector in direction of " + str(x1) + " is " \
        + str(unitvector(x1)) + ".")
```

If this file is executed using `python vectools.py`, then `__name__=="__main__"` is true, and the output reads

```
The norm of [0 1 2] is 2.2360679775 .
The unitvector in direction of [0 1 2] is [ 0. 0.4472136 0.89442719] .
```

If this file is imported (i.e. used as a module) into another python file, then `__name__=="__main__"` is false, and that statement block will not be executed (and no output produced).

This is quite a common way to conditionally execute code in files providing library-like functions. The code that is executed if the file is run on its own, often consists of a series of tests (to check that the file's functions carry out the right operations – *regression tests* or *unit tests*), or some examples of how the library functions in the file can be used.

7.5.5 Example 2

Even if a Python program is not intended to be used as a module file, it is good practice to always use a conditional main program:

- often, it turns out later that functions in the file can be reused (and saves work then)
- this is convenient for regression testing.

Suppose an exercise is given to write a function that returns the first 5 prime numbers, and in addition to print them. (There is of course a trivial solution to this as we know the prime numbers, and we should imagine that the required calculation is more complex). One might be tempted to write

```
def primes5():
    return (2, 3, 5, 7, 11)

for p in primes5():
    print "%d" % p,
```

It is better style to use a conditional main function, i.e.:

```
def primes5():
    return (2, 3, 5, 7, 11)

if __name__=="__main__":
    for p in primes5():
        print "%d" % p,
```

A purist might argue that the following is even cleaner:

```
def primes5():
    return (2, 3, 5, 7, 11)

def main():
    for p in primes5():
        print "%d" % p,

if __name__=="__main__":
    main()
```

but either of the last two options is good.

The example in section 9.1 demonstrates this technique. Including functions with names starting with `test_` is compatible with the very useful py.test regression testing framework (see http://pytest.org/).

Further Reading

- Python Tutorial Section 6

Chapter 8

Functional tools

Python provides a few in-built commands such as `map`, `filter`, `reduce` as well `lambda` (to create anonymous functions) and list comprehension. These are typical commands from functional languages of which LISP is probably best known.

Functional programming can be extremely powerful and one of the strengths of Python is that it allows to program using (i) imperative/procedural programming style, (ii) object oriented style and (iii) functional style. It is the programmers choice which tools to select from which style and how to mix them to best address a given problem.

In this chapter, we provide some examples for usage of the commands listed above.

8.1 Anonymous functions

All functions we have seen in Python so far have been defined through the `def` keyword, for example:

```
def f(x):
    return x ** 2
```

This funtion has the name `f`. Once the function is defined (i.e. the Python interpreter has come across the `def` line), we can call the function using its name, for example

```
y = f(x)
```

Sometimes, we need to define a function that is only used once, or we want to create a function but don't need a name for it (as for creating closures). In this case, this is called *anonymous* function as it does not have a name. In Python, the `lambda` keyword can create an anonymous function.

We create a (named) function first, check it's type and behaviour:

```
>>> def f(x):
...     return x ** 2
...
>>> f
<function f at 0x95a30>
>>> type(f)
<type 'function'>
>>> f(10)
100
```

Now we do the same with an anonymous function:

```
>>> lambda x: x ** 2
<function <lambda> at 0x957f0>
```

```
>>> type(lambda x: x ** 2)
<type 'function'>
>>> (lambda x: x ** 2)(10)
100
```

This works exactly in the same way but – as the anonymous function does not have a name – we need to define the function (through the `lambda` expression) – every time we need it.

Anonymous functions can take more than one argument:

```
>>> (lambda x, y: x + y)(10, 20)
30
>>> (lambda x, y, z: (x + y) * z )(10, 20, 2)
60
```

We will see some examples using `lambda` which will clarify typical use cases.

8.2 Map

The map function `lst2 = map(f, s )` applies a function `f` to all elements in a sequence `s`. The return value `lst2` of `map` is a list and has the same length as `s`:

```
>>> def f(x):
...     return x ** 2
>>> lst2 = map(f, range(10))
>>> lst2
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
>>> import string
>>> map( string.capitalize, ['banana', 'apple', 'orange'])
['Banana', 'Apple', 'Orange']
```

Often, this is combined with the anonymous function `lambda`:

```
>>> map( lambda x: x ** 2, range(10) )
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> map( lambda s: s.capitalize(), ['banana', 'apple', 'orange'])
['Banana', 'Apple', 'Orange']
```

8.3 Filter

The filter function `lst2 = filter( f, lst)` applies the function `f` to all elements in a sequence `s`. The function `f` should return `True` or `False`. The return value `lst2` is a list which will contain only those elements s_i of the sequence `s` for which `f`(s_i) has return `True`.

```
>>> def greater_than_5(x):
...     if x > 5:
...             return True
...     else:
...             return False
...
>>> filter(greater_than_5, range(11))
[6, 7, 8, 9, 10]
```

The usage of lambda can simplify this significantly:

```
>>> filter(lambda x: x > 5, range(11))
[6, 7, 8, 9, 10]
>>>
>>> known_names = ['smith', 'miller', 'bob']
>>> filter( lambda name : name in known_names, \
...         ['ago', 'smith', 'bob', 'carl'])
['smith', 'bob']
```

8.4 List comprehension

List comprehensions provide a concise way to create and modify lists without resorting to use of map(), filter() and/or lambda. The resulting list definition tends often to be clearer than lists built using those constructs. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the for and if clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized.

Some examples will make this clearer:

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3 * x for x in vec]
[6, 12, 18]
>>> [3 * x for x in vec if x > 3]
[12, 18]
>>> [3 * x for x in vec if x < 2]
[]
>>> [[x, x ** 2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

We can also use list comprehension to modify the list of integers returned by the range command so that our subsequent elements in the list increase by non-integer fractions:

```
>>> [x*0.5 for x in range(10)]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

Let's now revisit the examples from the section on filter

```
>>> [ x for x in range(11) if x>5 ]
[6, 7, 8, 9, 10]
>>>
>>> [ name for name in ['ago','smith','bob','carl'] \
...          if name in known_names]
['smith', 'bob']
```

and the examples from the map section

```
>>> [ x ** 2 for x in range(10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

```
>>> [ fruit.capitalize() for fruit in ['banana', 'apple', 'orange'] ]
['Banana', 'Apple', 'Orange']
```

all of which can be expressed through list comprehensions.

More details

- Python Tutorial 5.1.4 List comprehensions

8.5 Reduce

The `reduce` function takes a binary function `f(x,y)`, a sequence `s`, and a start value `a0`. It then applies the function `f` to the start value `a0` and the first element in the sequence: `a1 = f(a,s[0])`. The second element (`s[1]`) of the sequence is then processed as follows: the function f is called with arguments `a1` and `s[1]`, i.e. `a2 = f(a1,s[1])`. In this fashion, the whole sequence is processed. Reduce returns a single number.

This can be used, for example, to compute a sum of numbers in a sequence if the function `f(x,y)` returns `x+y`:

```
>>> def add(x,y):
...     return x+y
...
>>> reduce( add, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 0)
55
>>> reduce( add, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 100)
155
```

We can modify the function `add` to provide some more detail about the process:

```
>>> def add_verbose(x, y):
...     print "add(x=%s, y=%s) -> %s" % (x, y, x+y)
...     return x+y
...
>>> reduce( add_verbose, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 0)
add(x=0, y=1) -> 1
add(x=1, y=2) -> 3
add(x=3, y=3) -> 6
add(x=6, y=4) -> 10
add(x=10, y=5) -> 15
add(x=15, y=6) -> 21
add(x=21, y=7) -> 28
add(x=28, y=8) -> 36
add(x=36, y=9) -> 45
add(x=45, y=10) -> 55
55
```

It may be instructive to use an asymmetric function `f`, such as `add_len( n, s )` where s is a sequence and the function returns `n+len(s)` (suggestion from Thomas Fischbacher):

```
>>> def add_len(n,s):
...     return n+len(s)
...
>>> reduce( add_len, ["This","is","a","test."],0)
12
```

As before, we'll use a more verbose version of the binary function to see what is happening:

```
>>> def add_len_verbose(n,s):
...     print "add_len(n=%d, s=%s) -> %d" % (n, s, n+len(s))
...     return n+len(s)
...
>>> reduce( add_len_verbose, ["This", "is", "a", "test."], 0)
add_len(n=0, s=This) -> 4
add_len(n=4, s=is) -> 6
add_len(n=6, s=a) -> 7
add_len(n=7, s=test.) -> 12
12
>>>
```

Another way to understand what the reduce function does is to look at the following function (kindly provided by Thomas Fischbacher) which behaves like `reduce` but explains what it does:

```
def explain_reduce(f, xs, start=None):
    """This function behaves like reduce, but explains what it does,
    step-by-step.
    (Author: Thomas Fischbacher, modifications Hans Fangohr)"""
    nr_xs = len(xs)
    if start == None:
        if nr_xs == 0:
            raise ValueError("No starting value given - cannot " + \
                             "process empty list!")
        if nr_xs == 1:
            print("reducing over 1-element list without starting " + \
                  "value: returning that element.")
            return xs[0]
        else:
            print("reducing over list with >= 2 elements without " +\
                  "starting value: using the first element as a " +\
                  "start value.")
            return explain_reduce(f, xs[1:], xs[0])
    else:
        s = start
        for n in range(len(xs)):
            x = xs[n]
            print("Step %d: value-so-far=%s next-list-element=%s"\
                  % (n, str(s), str(x)))
            s = f(s, x)
        print("Done. Final result=%s" % str(s))
        return s
```

Here is an example using the `explain_reduce` function:

```
>>> from explain_reduce import explain_reduce
>>> def f(a,b):
...     return a+b
...
>>> reduce( f, [1,2,3,4,5], 0)
```

```
15
>>> explain_reduce( f, [1,2,3,4,5], 0)
Step 0: value-so-far=0 next-list-element=1
Step 1: value-so-far=1 next-list-element=2
Step 2: value-so-far=3 next-list-element=3
Step 3: value-so-far=6 next-list-element=4
Step 4: value-so-far=10 next-list-element=5
Done. Final result=15
15
```

Reduce is often combined with `lambda`:

```
>>> reduce( lambda x,y:x+y, [1,2,3,4,5], 0)
15
```

There is also the `operator` module which provides standard Python operators as functions. For example, the function `operator.__add__(a,b)` is executed when Python evaluates code such as `a+b`. These are generally faster than `lambda` expressions. We could write the example above as

```
>>> import operator
>>> reduce( operator.__add__, [1,2,3,4,5], 0)
15
```

Use `help('operator')` to see the complete list of operator functions.

8.6 Why not just use for-loops?

Let's compare the example introduced at the beginning of the chapter written (i) using a for-loop and (ii) list comprehension. Again, we want to compute the numbers $0^2, 1^2, 2^2, 3^2, \ldots$ up to $(n-1)^2$ for a given n.

Implementation (i) using a for-loop with n=10:

```
y = []
for i in range(10):
    y.append(i ** 2)
```

Implementation (ii) using list comprehension:

```
y = [ x ** 2 for x in range(10) ]
```

or using `map`:

```
y = map(lambda x: x ** 2, range(10))
```

The versions using list comprehension and `map` fit into one line of code whereas the for-loop needs 3. This example shows that functional code result in very *concise* expressions. Typically, the number of mistakes a programmer makes is per line of code written, so the fewer lines of code we have, the fewer bugs we need to find.

Often programmers find that initially the list-processing tools introduced in this chapter seem less intuitive than using for-loops to process every element in a list individually, but that – over time – they come to value a more functional programming style.

8.7 Speed

The functional tools described in this chapter can also be faster than using explicit (for or while) loops over list elements.

The program `list_comprehension_speed.py` below computes $\sum_{i=0}^{N-1} i^2$ for a large value of N using 4 different methods and records execution time:

- Method 1: for-loop (with pre-allocated list, storing of i^2 in list, then using in-built `sum` function)
- Method 2: for-loop without list (updating sum as the for-loop progresses)
- Method 3: using list comprehension
- Method 4: using numpy. (Numpy is covered in section 14.1)

```
"""Compare calculation of \sum_i x_i^2) with
i going from zero to N-1.

We use (i) for loops and list, (ii) for-loop, (iii) list comprehension
and (iv) numpy.

We use floating numbers to avoid using Python's long int (which would
be likely to make the timings less representative).
"""

import time
import numpy
N = 10000000


def timeit(f, args):
    """Given a function f and a tuple args containing
    the arguments for f, this function calls f(*args),
    and measures and returns the execution time in
    seconds.

    Return value is tuple: entry 0 is the time,
    entry 1 is the return value of f."""

    starttime = time.time()
    y = f(*args)     # use tuple args as input arguments
    endtime = time.time()
    return endtime - starttime, y


def forloop1(N):
    s = 0
    for i in range(N):
        s += float(i) * float(i)
    return s
```

```
def forloop2(N):
    y = [0] * N
    for i in range(N):
        y[i] = float(i) ** 2
    return sum(y)


def listcomp(N):
    return sum([float(x) * x for x in range(N)])


def numpy_(N):
    return numpy.sum(numpy.arange(0, N, dtype='d') ** 2)

#main program starts
print "N =", N
forloop1_time, f1_res = timeit(forloop1, (N,))
print "for-loop1", forloop1_time
forloop2_time, f2_res = timeit(forloop2, (N,))
print "for-loop2", forloop2_time
listcomp_time, lc_res = timeit(listcomp, (N,))
print "listcomp", listcomp_time
numpy_time, n_res = timeit(numpy_, (N,))
print "numpy", numpy_time

#ensure that different methods provide identical results
assert f1_res == f2_res
assert f1_res == lc_res
assert f1_res == n_res
```

The program produces the following output:

```
N = 10000000
for-loop1 7.45045185089
for-loop2 5.96237707138
listcomp 4.66884493828
numpy 0.262656927109
```

We see that methods 1 (for-loop with list) is slowest with about 7.5 seconds, method 2 (for-loop without list) is faster with 6 seconds. Method 3 (list comprehension) is faster using only 4.7 seconds. The faster method is 4 (using numpy) which only requires 0.25 seconds, and is thus about 30 times faster than method 1.

Chapter 9

Common tasks

Here we provide a selection of small example programs addressing some common tasks and just providing some more Python code that can be read if seeking inspiration how to address a given problem.

9.1 Many ways to compute a series

As an example, we compute the sum of odd numbers in different ways.

```
def compute_sum1(n):
    """computes and returns the sum of 2,4,6, ..., m
    where m is the largest even number smaller than n.

    For example, with n = 7, we compute 0+2+4+6 = 12.

    This implementation uses a variable 'mysum' that is
    increased in every iteration of the for-loop."""

    mysum = 0
    for i in range(0, n, 2):
        mysum = mysum + i
    return mysum


def compute_sum2(n):
    """computes and returns ...

    This implementation uses a while-loop:
    """

    counter = 0
    mysum = 0
    while counter < n:
        mysum = mysum + counter
        counter = counter + 2

    return mysum
```

```
def compute_sum3(n, startfrom=0):
    """computes and returns ...

    This is a recursive implementation:"""

    if n <= startfrom:
        return 0
    else:
        return startfrom + compute_sum3(n, startfrom + 2)


def compute_sum4a(n):
    """A functional approach ... this seems to be
    the shortest and most concise code.
    """
    return sum(range(0, n, 2))


def compute_sum4b(n):
    """A functional approach ... not making use of 'sum' which
    happens to exist and is of course convenient here.
    """
    return reduce(lambda a, b: a + b, range(0, n, 2))


def compute_sum4c(n):
    """A functional approach ... a bit faster than compute_sum4b
    as we avoid using lambda.
    """
    import operator
    return reduce(operator.__add__, range(0, n, 2))


def compute_sum4d(n):
    """Using list comprehension."""
    return sum([k for k in range(0, n, 2)])


def compute_sum4e(n):
    """Using another variation of list comprehension."""
    return sum([k for k in range(0, n) if k % 2 == 0])


def compute_sum5(n):
    """Using numerical python (numpy). This is very fast
    (but would only pay off if n >> 10)."""
    import numpy
    return numpy.sum(2 * numpy.arange(0, (n + 1) // 2))
```

```
def test_consistency():
    """Check that all compute_sum?? functions in this file produce
    the same answer for all n>=2 and <N.
    """
    def check_one_n(n):
        """Compare the output of compute_sum1 with all other functions
        for a given n>=2. Raise AssertionError if outputs disagree."""
        funcs = [compute_sum1, compute_sum2, compute_sum3,
                 compute_sum4a, compute_sum4b, compute_sum4c,
                 compute_sum4d, compute_sum4e, compute_sum5]
        ans1 = compute_sum1(n)
        for f in funcs[1:]:
            assert ans1 == f(n), "%s(n)=%d not the same as %s(n)=%d " \
                                 % (funcs[0], funcs[0](n), f, f(n))

    #main testing loop in test_consistency function
    for n in range(2, 1000):
        check_one_n(n)

if __name__ == "__main__":
    m = 7
    correct_result = 12
    thisresult = compute_sum1(m)
    print("this result is {}, expected to be {}".format(
        thisresult, correct_result))
    # compare with correct result
    assert thisresult == correct_result
    # also check all other methods
    assert compute_sum2(m) == correct_result
    assert compute_sum3(m) == correct_result
    assert compute_sum4a(m) == correct_result
    assert compute_sum4b(m) == correct_result
    assert compute_sum4c(m) == correct_result
    assert compute_sum4d(m) == correct_result
    assert compute_sum4e(m) == correct_result
    assert compute_sum5(m) == correct_result

    # a more systematic check for many values
    test_consistency()
```

Running this program produces the following output:

```
this result is 12, expected to be 12
```

All the different implementations shown above compute the same result. There are a number of things to be learned from this:

- There are a large (probably an infinite) number of solutions for one given problem. (This means that writing programs is a task that requires creativity!)
- These may achieve the same 'result' (in this case computation of a number).

- Different solutions may have different characteristics. They might:
  - ▷ be faster or slower
  - ▷ use less or more memory
  - ▷ are easier or more difficult to understand (when reading the source code)
  - ▷ can be considered more or less elegant.

9.2 Sorting

Suppose we need to sort a list of 2-tuples of user-ids and names, i.e.

```
mylist = [("fangohr", "Hans Fangohr",),
          ("admin", "The Administrator"),
          ("guest", "The Guest")]
```

which we want to sort in increasing order of user-ids. If there are two or more identical user-ids, they should be ordered by the order of the names associated with these user-ids. This behaviour is just the default behaviour of sort (which goes back to how to sequences are compared).

```
stuff = mylist # collect your data
stuff.sort()   # sort the data in place
print stuff    # inspect the sorted data
```

Sequences are compared by initially comparing the first elements only. If they differ, then a decision is reached on the basis of those elements only. If the elements are equal, only then are the next elements in the sequence compared ... and so on, until a difference is found, or we run out of elements. For example:

```
>>> (2,0) > (1,0)
True
>>> (2,1) > (1,3)
True
>>> (2,1) > (2,1)
False
>>> (2,2) > (2,1)
True
```

From Python 2.4 onwards, it is also possible to do

```
stuff = sorted(stuff)
```

Where the list is not particularly large, it is generally advisable to use the `sorted` function (which *returns a sorted copy* of the list) over the `sort` method of a list (which changes the list into sorted order of elements, and returns None).

However, what if the data we have is stored such that in each tuple in the list, the name comes first, followed by the id, i.e.:

```
mylist2 = [("Hans Fangohr", "fangohr"),
           ("The Administrator", "admin"),
           ("The Guest", "guest")]
```

We want to sort with the id as the primary key. The first approach to do this is to change the order of `mylist2` to that of `mylist`, and use `sort` as shown above.

The second approach (which also works in ancient Pythons), relies on being able to decypher the cryptic `help(list.sort)`. You should notice that list.sort has three keyword parameters. The first of these is called cmp. If this argument is None (which it will be, by default), `list.sort` will use Python's built-in cmp function to compare the elements of the list it is sorting. `cmp` is a function of two arguments which returns a positive number when it considers the first argument to be 'greater' then the second, returns 0 when it considers the arguments to be equal, and returns a negative number otherwise. You can instruct `list.sort` to use a different comparison criterion, by supplying some other function with the same interface.

Let's illustrate this in the context of our exercise, by assuming that we have stored a list of pairs like this

```
pair = name, id
```

(i.e. as in `mylist2`) and that we want to sort according to id and ignore name. We can achieve this by writing a function that compares the second elements of the pairs it receives, and returns a number matching the convention used by cmp as described above.

```
def my_cmp(a, b):
    if a[1] > b[1]:
        return +1
    if a[1] == b[1]:
        return 0
    return -1
```

Note that you must remember the convention, and that there is a lot of fiddling around with indices. A much more concise and less error-prone (and more efficient) way of writing this function is

```
def my_cmp(a, b):
    return cmp(a[1], b[1])
```

Now you can use this function to tell `list.sort` how to compare the pairs:

```
pairs.sort(cmp=my_cmp)
```

or

```
pairs.sort(my_cmp)
```

or even

```
pairs.sort(lambda a, b: cmp(a[1], b[1]))
```

Another solution, which became available in Python 2.4, relies on `cmp`'s keyword parameter key. Here you can pass a function which `list.sort` will use to generate values according to which the list's elements should be sorted. In our case

```
pairs.sort(key=lambda p: p[1])
```

will do the trick.

A note about efficiency. Sorting typically involves making many comparisons between the elements. Python's built-in cmp is an efficient low-level function. By making `list.sort` use a pure-Python function instead, you are getting hit by the pure-Python function overhead (which is relatively large compared to the C function overhead) on *each and every comparison*. This might slow you down noticeably.

The `key` function will be called exactly once for every element in the list. This approach is likely to be measurably faster for large lists than using the `cmp` approach.

If efficiency is really important (and you have proven that a significant proportion of time is spent in these functions) then you have the option of re-coding them in C (or another low-level language).

Chapter 10

From Matlab to Python

10.1 Important commands

10.1.1 The for-loop

Matlab

```
for i = 1:10
    disp(i)
end
```

Matlab requires the end key-word at the end of the block belonging to the for-loop.

Python

```
for i in range(1,11):
    print(i)
```

Python requires a colon (":") at the of the for-line. (This is important and often forgotten when you have programmed in Matlab before.) Python requires the commands to be executed within the for-loop to be indented.

10.1.2 The if-then statement

Matlab

```
if a==0
    disp('a is zero')
elseif a<0
    disp('a is negative')
elseif a==42
    disp('a is 42')
else
    disp('a is positive')
end
```

Matlab requires the end key-word at the very end of the block belonging to the for-loop.

Python

```
if a==0:
    print('a is zero')
elif a<0:
    print('a is negative')
elif a==42:
    print('a is 42')
else:
    print('a is positive')
```

Python requires a colon (":") after every condition (i.e. at the of the lines starting with if, elif, else. Python requires the commands to be executed within each part of the if-then-else statement to be indented.

10.1.3 Indexing

Matlab's indexing of matrices and vectors starts a 1 (similar to Fortran), whereas Python's indexing starts at 0 (similar to C).

10.1.4 Matrices

In Matlab, every object is a matrix. In Python, there is a specialised extension library called `numpy` (see Sec. 14) which provides the `array` object which in turns provides the corresponding functionality. Similar to Matlab, the `numpy` object is actually based on binary libraries and execution there very fast.

There is a dedicated introduction to numpy for Matlab users available at http://www.scipy.org/NumPy_for_Matlab_Users.

Chapter 11

Python shells

11.1 IDLE

IDLE comes with every Python distribution and is a useful tool for everyday programming. Its editor provides syntax highlighting.

There are two reasons why you might want to use another Python shell, for example:

- While working with the Python prompt, you like auto-completion of variable names, filenames and commands. In that case "Ipython" is your tool of choice (see section 11.3.1). Ipython does not provide an editor but you can carry on using the IDLE editor to edit files, or any other editor you like.

  IPython provides a number of nice features for the more experienced Python programmer, including convenient profiling of code (see http://ipython.scipy.org).

  Recently, some auto-completion has been added to Idle as well (press tab after having typed the first few letters of object names and keywords).

11.2 Python (command line)

This is the most basic face of the Python shell. It is very similar to the Python prompt in IDLE but there are no menus to click on and no facilities to edit files.

11.3 Interactive Python (IPython)

11.3.1 IPython console

IPython is an improved version of the Python command line. It is a valuable tool and worth exploring it's capabilities (see http://ipython.org/ipython-doc/stable/interactive/qtconsole.html)

You will find the following features very useful:

- auto completion
  Suppose you want to type `a = range(10)`. Instead of typing all the letters, just type `a = ra` and the press the "Tab" key. Ipython will now show all the possible commands (and variable names) that start with `ra`. If you type a third letter, here `n` and press "Tab" again, Ipython will auto complete and append `ge` automatically.
  This works also for variable names and modules.

- To obtain help on a command, we can use Python's help command. For example: `help(range)`. Ipython provides a shortcut. To achieve the same, it is sufficient to just type the command followed by a question mark: `range?`

- You can relatively easily navigate directories on your computer. For example,
  - ▷ `!dir` lists the content of the current directory (same as `ls`)
  - ▷ `pwd` shows the current working directory
  - ▷ `cd` allows to change directories

  In general, using an exclamation mark before the command will pass the command to the shell (not to the Python interpreter).

- You can execute Python programs from ipython using `%run`. Suppose you have a file `hello.py` in the current directory. You can then execute it by typing: `%run hello`

  Note that this differs from executing a python program in IDLE: IDLE restarts the Python interpreter session and thus deletes all existing objects before the execution starts. This is not the case with the `run` command in ipython (and neither when executing chunks of Python code from Emacs using the Emacs Python mode). In particular this can be very useful if one needs to setup a few objects which are needed to test the code one is working on. Using ipython's `run` or Emacs instead of IDLE allows to keep these objects in the interpreter session and to only update the function/classes/... etc that are being developed.

- allows multi-line editing of command history
- provides on-the-fly syntax highlighting
- displays doc-strings on-the-fly
- can inline matplotlib figures (activate mode with if started with `%matplotlib inline`)
- `%load` loads file from disk or form URL for editing
- `%timeit` measures execution time for a given statement
- ...and a lot more.
- Read more at http://ipython.org/ipython-doc/dev/interactive/qtconsole.html

If you have access to this shell, you may want to consider it as your default Python prompt.

11.3.2 IPython Notebook

The IPython Notebook allows to execute, store, load, re-execute a sequence of Python commands, and to include explanatory text, images and other media in between.

This is a recent and exciting development that has the potential to develop into a tool of great significance, for example for

- documenting calculations and data processing
- support learning and teaching of
  - ▷ Python itself
  - ▷ statistical methods
  - ▷ general data post-processing
  - ▷ ...
- documentation new code

- automatic regression testing by re-running ipython notebook and comparing stored output with computed output

Further reading

- IPython notebook (http://ipython.org/notebook.html).
- Further materials at http://ipython.org.

11.4 Spyder

Spyder is the Scientific PYthon Development EnviRonment: a powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging and introspection features and a numerical computing environment thanks to the support of IPython (enhanced interactive Python interpreter) and popular Python libraries such as NumPy (linear algebra), SciPy (signal and image processing) or matplotlib (interactive 2D/3D plotting). See http://pythonhosted.org/spyder/ for more.

Some important features of Spyder:

- Within Spyder, the IPython console is the default Python interpreter, and
- code in the editor can be fully or partially be executed in this buffer.
- The editor supports automatic checking for Python erros using pyflakes, and
- the editor warns (if desired) if the code formatting deviates from the PEP8 style guide.
- The Ipython Debugger can be activated, and
- a profiler is provided.
- An object explorer shows documentation for functions, methods etc on the fly and a
- variable explorer displays names, size and values for numerical variables.

Spyder is currently (as of 2014) on the way to develop into a powerful and robust multi-platform integrated environment for Python development, with particular emphasis on Python for scientific computing and engineering.

11.5 Editors

All major editors that are used for programming, provide Python modes (such as Emacs, Vim, Sublime Text), some Integrated Development Enviroments (IDEs) come with their own editor (Spyder, Eclipse). Which of these is best, is partly a matter of choice.

For beginners, Spyder seems a sensible choice as it provides an IDE, allows execution of chunks of code in an interpreter session and is easy to pick up.

Chapter 12

Symbolic computation

12.1 SymPy

In this section, we introduce some basic functionality of the SymPy (SYMbolic Python) library. In contrast to numerical computation (involving numbers), in symbolic calculation we are processing and transforming generic variables.

The SymPy home page is http://sympy.org/, and provides the full (and up-to-date) documentation for this library.

Symbolic calculation is very slow compared to floating point operation (see for example Sec. 13.1.5), and thus generally not for direct simulation. However, it is a powerful tool to support the preparation of code and symbolic work. Occasionally, we use symbolic operations in simulations to work out the most efficient numerical code, before that is executed.

12.1.1 Symbols

Before we can carry out any symbolic operations, we need to create symbolic variables using SymPy's `Symbol` function:

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> type(x)
<class 'sympy.core.symbol.Symbol'>
>>> y = Symbol('y')
>>> 2 * x - x
x
>>> x + y + x + 10*y
2*x + 11*y
>>> y + x - y + 10
10 + x
```

We can abbreviate the creation of multiple symbolic variables using the `symbols` function. For example, to create the symbolic variables `x`, `y` and `z`, we can use

```
>>> import sympy
>>> x, y, z = sympy.symbols('x,y,z')
>>> x + 2*y + 3*z - x
2*y + 3*z
```

Once we have completed our term manipulation, we sometimes like to insert numbers for variables. This can be done using the `subs` method.

```
>>> from sympy import symbols
>>> x, y = symbols('x,y')
>>> x + 2*y
x + 2*y
>>> x + 2*y.subs(x, 10)
x + 2*y
>>> (x + 2*y).subs(x, 10)
10 + 2*y
>>> (x + 2*y).subs(x, 10).subs(y, 3)
16
>>> (x + 2*y).subs({x:10, y:3})
16
```

We can also substitute a symbolic variable for another one such as in this example where `y` is replaced with `x` before we substitute `x` with the number `2`.

```
>>> myterm = 3*x + y**2
>>> myterm
3*x + y**2
>>> myterm.subs(x, y)
3*y + y**2
>>> myterm.subs(x, y).subs(y, 2)
10
```

From this point onward, some of the code fragments and examples we present will assume that the required symbols have already been defined. If you try an example and SymPy gives a message like `NameError: name 'x' is not defined` it is probably because you need to define the symbol using one of the methods above.

12.1.2 isympy

The `isympy` executable is a wrapper around ipython (see section 11.3.1) which creates the symbolic (real) variables `x`, `y` and `z`, the symbolic integer variables `k`, `m` and `n` and the symbolic function variables `f`, `g` and `h`, and imports all objects from the SymPy toplevel.

This is convenient to figure out new features or experimenting interactively

```
$> isympy
Python 2.6.5 console for SymPy 0.6.7

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z = symbols('xyz')
>>> k, m, n = symbols('kmn', integer=True)
>>> f, g, h = map(Function, 'fgh')

Documentation can be found at http://sympy.org/

In [1]:
```

12.1.3 Numeric types

SymPy has the numeric types `Rational` and `Real`. The Rational class represents a rational number as a pair of two integers: the numerator and the denominator, so `Rational(1,2)` represents `1/2`, `Rational(5,2)` represents `5/2` and so on.

```
>>> a = Rational(1, 10)
>>> a
1/10
>>> b = Rational(45, 67)
>>> b
45/67
>>> a * b
9/134
>>> a - b
-383/670
>>> a + b
517/670
```

Note that the Rational class works with rational expressions *exactly*. This is in contrast to Python's standard `float` data type which uses floating point representation to *approximate* (rational) numbers.

We can convert the `sympy.Rational` type into a Python floating point variable using `float` or the `evalf` method of the Rational object. The `evalf` method can take an argument that specifies how many digits should be computed for the floating point approximation (not all of those may be used by Python's floating point type of course).

```
>>> c = Rational(2, 3)
>>> c
2/3
>>> float(c)
0.66666666666666663
>>> c.evalf()
0.666666666666667
>>> c.evalf(50)
0.66666666666666666666666666666666666666666666666667
```

12.1.4 Differentiation and Integration

SymPy is capable of carrying out differentiation and integration of many functions:

```
>>> from sympy import Symbol, exp, sin, sqrt, diff
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> diff(sin(x), x)
cos(x)
>>> diff(sin(x), y)
0
>>> diff(10 + 3*x + 4*y + 10*x**2 + x**9, x)
3 + 20*x + 9*x**8
>>> diff(10 + 3*x + 4*y + 10*x**2 + x**9, y)
4
>>> diff(10 + 3*x + 4*y + 10*x**2 + x**9, x).subs(x,1)
```

```
32
>>> diff(10 + 3*x + 4*y + 10*x**2 + x**9, x).subs(x,1.5)
263.660156250000
>>> diff(exp(x), x)
exp(x)
>>> diff(exp(-x ** 2 / 2), x)
-x*exp(-x**2/2)
```

The SymPy `diff()` function takes a minimum of two arguments: the function to be differentiated and the variable with respect to which the differentiation is performed. Higher derivatives may be calculated by specifying additional variables, or by adding an optional integer argument:

```
>>> diff(3*x**4, x)
12*x**3
>>> diff(3*x**4, x, x, x)
72*x
>>> diff(3*x**4, x, 3)
72*x
>>> diff(3*x**4*y**7, x, 2, y, 2)
1512*x**2*y**5
>>> diff(diff(3*x**4*y**7, x, x), y, y)
1512*x**2*y**5
```

At times, SymPy may return a result in an unfamiliar form. If, for example, you wish to use SymPy to check that you differentiated something correctly, a technique that might be of use is to subtract the SymPy result from your result, and check that the answer is zero.

Taking the simple example of a multiquadric radial basis function, $\phi(r) = \sqrt{r^2 + \sigma^2}$ with $r = \sqrt{x^2 + y^2}$ and σ a constant, we can verify that the first derivative in x is $\partial\phi/\partial x = x/\sqrt{r^2 + \sigma^2}$. In this example, we first ask SymPy to print the derivative. See that it is printed in a different form to our trial derivative, but the subtraction verifies that they are identical:

```
>>> r = sqrt(x**2 + y**2)
>>> def phi(x,y,sigma):
...     return sqrt(x**2 + y**2 + sigma**2)
...
>>> mydfdx= x / sqrt(r**2 + sigma**2)
>>> print diff(phi(x, y, sigma), x)
x/(sigma**2 + x**2 + y**2)**(1/2)
>>> print mydfdx - diff(phi(x, y, sigma), x)
0
```

Here it is trivial to tell that the expressions are identical without SymPy's help, but in more complicated examples there may be many more terms and it would become increasingly difficult, time consuming and error-prone to attempt to rearrange our trial derivative and SymPy's answer into the same form. It is in such cases that this subtraction technique is of most use.

Integration uses a similar syntax. For the indefinite case, specify the function and the variable with respect to which the integration is performed:

```
>>> from sympy import integrate
>>> integrate(x**2, x)
x**3/3
>>> integrate(x**2, y)
y*x**2
```

```
>>> integrate(sin(x), y)
y*sin(x)
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(-x*exp(-x**2/2), x)
exp(-x**2/2)
```

We can calculate definite integrals by providing `integrate()` with a tuple containing the variable of interest, the lower and the upper bounds. If several variables are specified, multiple integration is performed. When SymPy returns a result in the `Rational` class, it is possible to evaluate it to a floating-point representation at any desired precision (see section 12.1.3).

```
>>> integrate(x*2, (x, 0, 1))
1
>>> integrate(x**2, x)
x**3/3
>>> integrate(x**2, x, x)
x**4/12
>>> integrate(x**2, x, x, y)
y*x**4/12
>>> integrate(x**2, (x, 0, 2))
8/3
>>> integrate(x**2, (x, 0, 2), (x, 0, 2), (y, 0, 1))
16/3
>>> float(integrate(x**2, (x, 0, 2)))
2.6666666666666665
>>> type(integrate(x**2, (x, 0, 2)))
<class 'sympy.core.numbers.Rational'>
>>> result_rational=integrate(x**2, (x, 0, 2))
>>> result_rational.evalf()
2.66666666666667
>>> result_rational.evalf(50)
2.6666666666666666666666666666666666666666666666667
```

12.1.5 Ordinary differential equations

SymPy has inbuilt support for solving several kinds of ordinary differential equation via its `dsolve` command. We need to set up the ODE and pass it as the first argument, `eq`. The second argument is the function `f(x)` to solve for. An optional third argument, `hint`, influences the method that `dsolve` uses: some methods are better-suited to certain classes of ODEs, or will express the solution more simply, than others.

To set up the ODE solver, we need a way to refer to the unknown function for which we are solving, as well as its derivatives. The `Function` and `Derivative` classes facilitate this:

```
>>> from sympy import Symbol, dsolve, Function, Derivative, Eq
>>> y = Function("y")
>>> x = Symbol('x')
>>> y_ = Derivative(y(x), x)
>>> print dsolve(y_ + 5*y(x), y(x))
y(x) == exp(C1 - 5*x)
```

Note how dsolve has introduced a constant of integration, C1. It will introduce as many constants as are required, and they will all be named Cn, where n is an integer. Note also that the first argument to dsolve is taken to be equal to zero unless we use the Eq() function to specify otherwise:

```
>>> print dsolve(y_ + 5*y(x), y(x))
y(x) == exp(C1 - 5*x)
>>> print dsolve(Eq(y_ + 5*y(x), 0), y(x))
y(x) == exp(C1 - 5*x)
>>> print dsolve(Eq(y_ + 5*y(x), 12), y(x))
y(x) == 12/5 - exp(C1 - 5*x)/5
```

The results from dsolve are an instance of the Equality class. This has consequences when we wish to numerically evaluate the function and use the result elsewhere (*e.g.* if we wanted to plot $y(x)$ against x), because even after using subs() and evalf(), we still have an Equality, not any sort of scalar. The way to evaluate the function to a number is via the rhs attribute of the Equality.

Note that, here, we use z to store the Equality returned by dsolve, even though it is an expression for a function called y(x), to emphasise the distinction between the Equality itself and the data that it contains.

```
>>> z = dsolve(y_ + 5*y(x), y(x))
>>> print z
y(x) == exp(C1 - 5*x)
>>> type(z)
<class 'sympy.core.relational.Equality'>
>>> print z.rhs
exp(C1 - 5*x)
>>> C1=Symbol('C1')
>>> y3 = z.subs({C1:2, x:3})
>>> print y3
y(3) == exp(-13)
>>> y3.evalf(10)
y(3) == exp(-13)
>>> y3.rhs
exp(-13)
>>> y3.rhs.evalf(10)
2.260329407e-6
>>> z.rhs.subs({C1:2, x:4}).evalf(10)
1.522997974e-8
>>> z.rhs.subs({C1:2, x:5}).evalf(10)
1.026187963e-10
>>> type(z.rhs.subs({C1:2, x:5}).evalf(10))
<class 'sympy.core.numbers.Real'>
```

At times, dsolve may return too general a solution. One example is when there is a possibility that some coefficients may be complex. If we know that, for example, they are always real and positive, we can provide dsolve this information to avoid the solution becoming unnecessarily complicated:

```
>>> from sympy import *
>>> a, x = symbols('a,x')
>>> f = Function('f')
>>> print dsolve(Derivative(f(x), x, 2) + a**4*f(x), f(x))
f(x) == (C1*sin(x*abs(im((-a**4)**(1/2)))) + C2*cos(x*im((-a**4)
   **(1/2))))*exp(x*re((-a**4)**(1/2))) + (C3*sin(x*abs(im(-(-a**4)
```

```
   **(1/2)))) + C4*cos(x*im(-(-a**4)**(1/2))))*exp(x*re(-(-a**4)
   **(1/2)))
>>> a=Symbol('a',real=True,positive=True)
>>> print dsolve(Derivative(f(x), x, 2)+a**4*f(x), f(x))
f(x) == C1*sin(x*a**2) + C2*cos(x*a**2)
```

12.1.6 Series expansions and plotting

It is possible to expand many SymPy expressions as Taylor series. The `series` method makes this straightforward. At minimum, we must specify the expression and the variable in which to expand it. Optionally, we can also specify the point around which to expand, the maximum term number, and the direction of the expansion (try `help(Basic.series)` for more information).

```
>>> from sympy import *
>>> x = Symbol('x')
>>> sin(x).series(x, 0)
x - x**3/6 + x**5/120 + O(x**6)
>>> series(sin(x), x, 0)
x - x**3/6 + x**5/120 + O(x**6)
>>> cos(x).series(x, 0.5, 10)
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)
```

In some cases, especially for numerical evaluation and plotting the results, it is necessary to remove the trailing `O(n)` term:

```
>>> cos(x).series(x, 0.5, 10).removeO()
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320
```

SymPy provides two inbuilt plotting functions, `Plot()` from the `sympy.plotting` module, and `plot` from `sympy.mpmath.visualization`. At the time of writing, these functions lack the ability to add a key to the plot, which means they are unsuitable for most of our needs. Should you wish to use them nevertheless, their `help()` text is useful.

For most of our purposes, Matplotlib should be the plotting tool of choice. The details are in chapter 15. Here we furnish just one example of how to plot the results of a SymPy computation. The resulting plot is shown in figure 12.1.

```
>>> from sympy import sin,series,Symbol
>>> import pylab
>>> x = Symbol('x')
>>> s10 = sin(x).series(x,0,10).removeO()
>>> s20 = sin(x).series(x,0,20).removeO()
>>> s = sin(x)
>>> xx = []
>>> y10 = []
>>> y20 = []
>>> y = []
>>> for i in range(1000):
...   xx.append(i / 100.0)
...   y10.append(float(s10.subs({x:i/100.0})))
...   y20.append(float(s20.subs({x:i/100.0})))
...   y.append(float(s.subs({x:i/100.0})))
...
```

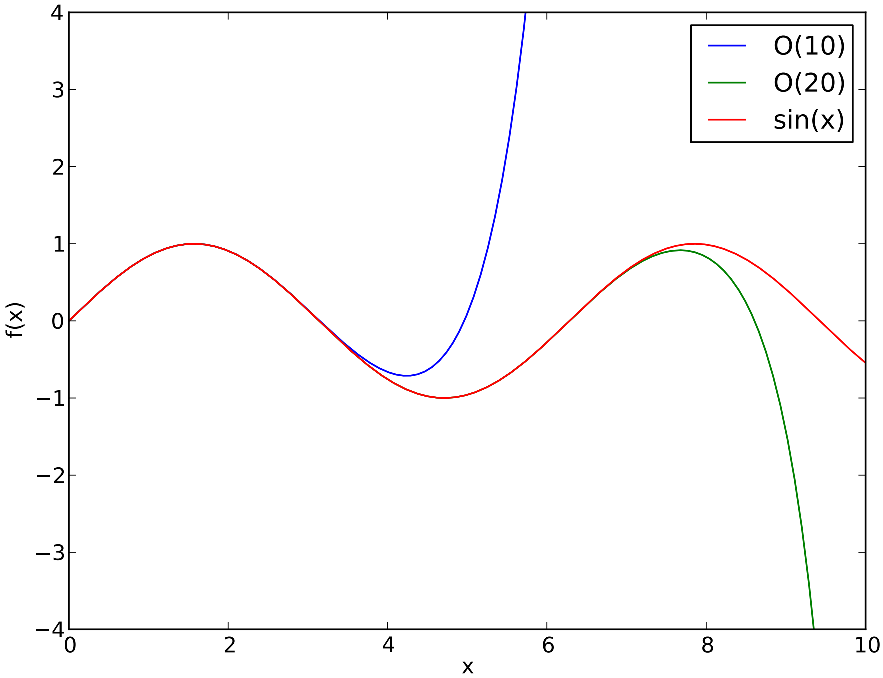


Figure 12.1: Plotting SymPy results with Matplotlib.

```
>>> pylab.figure()
<matplotlib.figure.Figure object at 0x3f45e10>
>>> pylab.plot(xx, y10, label='O(10)')
[<matplotlib.lines.Line2D object at 0x44a8d10>]
>>> pylab.plot(xx, y20, label='O(20)')
[<matplotlib.lines.Line2D object at 0x44b6750>]
>>> pylab.plot(xx, y, label='sin(x)')
[<matplotlib.lines.Line2D object at 0x3f4eb90>]
>>> pylab.axis([0, 10, -4, 4])
[0, 10, -4, 4]
>>> pylab.xlabel('x')
<matplotlib.text.Text object at 0x4374490>
>>> pylab.ylabel('f(x)')
<matplotlib.text.Text object at 0x43698d0>
>>> pylab.legend()
<matplotlib.legend.Legend object at 0x449c710>
>>> pylab.savefig('sympy.pdf')
>>> pylab.savefig('sympy.png')
>>> pylab.show()
```

12.1.7 Linear equations and matrix inversion

SymPy has a `Matrix` class and associated functions that allow the symbolic solution of systems of linear equations (and, of course, we can obtain numerical answers with `subs()` and `evalf()`). We

shall consider the example of the following simple pair of linear equations:

$$3x + 7y = 12z$$
$$4x - 2y = 5z$$

We may write this system in the form $A\vec{x} = \vec{b}$ (multiply A by $\vec{x}$ if you want to verify that we recover the original equations), where

$$A = \begin{pmatrix} 3 & 7 \\ 4 & -2 \end{pmatrix}, \qquad \vec{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} 12z \\ 5z \end{pmatrix}.$$

Here we included a symbol, z, on the right-hand side to demonstrate that symbols will be propagated into the solution. In many cases we would have $z = 1$, but there may still be benefit to using SymPy over a numerical solver even when the solution contains no symbols because of its ability to return exact fractions rather than approximate `floats`.

One strategy to solve for $\vec{x}$ is to invert the matrix A and pre-multiply, *i.e.* $A^{-1}A\vec{x} = \vec{x} = A^{-1}\vec{b}$. SymPy's `Matrix` class has an `inv()` method that allows us to find the inverse, and `*` performs matrix multiplication for us, when appropriate:

```
>>> from sympy import symbols,Matrix
>>> x, y, z = symbols('xyz')
>>> A = Matrix(([3, 7], [4, -2]))
>>> print A
[3,  7]
[4, -2]
>>> print A.inv()
[1/17,  7/34]
[2/17, -3/34]
>>> b = Matrix(( 12*z,5*z  ))
>>> print b
[12*z]
[ 5*z]
>>> x = A.inv()*b
>>> print x
[59*z/34]
[33*z/34]
>>> print x.subs({z:3.3}).evalf(4)
[5.726]
[3.203]
>>> type(x)
<class 'sympy.matrices.matrices.Matrix'>
```

An alternative method of solving the same problem is to construct the system as a matrix in augmented form; that is the form obtained by appending the columns of (in our example) A and $\vec{b}$ together. The augmented matrix is[1]:

$$(A|\vec{b}) = \left(\begin{array}{cc|c} 3 & 7 & 12z \\ 4 & -2 & 5z \end{array}\right),$$

and as before we construct this as a SymPy `Matrix` object, but in this case we pass it to the `solve_linear_system()` function:

[1] the vertical line is to show the division between the original components only; mathematically, the augmented matrix behaves like any other 2×3 matrix, and we code it in SymPy as we would any other.

```
>>> from sympy import Matrix, symbols, solve_linear_system
>>> x, y, z = symbols('xyz')
>>> system = Matrix(([3, 7, 12*z],[4, -2, 5*z]))
>>> print system
[3,  7, 12*z]
[4, -2,  5*z]
>>> sol = solve_linear_system(system,x,y)
>>> print sol
{x: 59*z/34, y: 33*z/34}
>>> type(sol)
<type 'dict'>
>>> for k in sol.keys():
...     print k,'=',sol[k].subs({z:3.3}).evalf(4)
...
x = 5.726
y = 3.203
```

A third option is the `solve()` method, whose arguments include the individual symbolic equations, rather than any matrices. Like `dsolve()` (see section 12.1.5), `solve()` expects either expressions which it will assume equal to zero, or `Equality` objects, which we can conveniently create with `Eq()`:

```
>>> from sympy import symbols,solve,Eq
>>> x, y, z = symbols('xyz')
>>> solve((Eq(3*x+7*y,12*z), Eq(4*x-2*y,5*z)), x, y)
{x: 59*z/34, y: 33*z/34}
>>> solve((3*x+7*y-12*z, 4*x-2*y-5*z), x, y)
{x: 59*z/34, y: 33*z/34}
```

For more information, see `help(solve)` and `help(solve_linear_system)`.

12.1.8 Non linear equations

Let's solve a simple equation such as

$$x = x^2$$

. There are two obvious solutions: $x = 0$ and $x = 1$. How can we ask Sympy to compute these for us?

```
>>> import sympy
>>> x, y, z = sympy.symbols('x, y, z')        # create some symbols
>>> eq = x - x ** 2                           # define the equation

>>> sympy.solve(eq, x)                        # solve eq = 0
[0, 1]                                        # this is the solution
```

The `solve()` function expects an expression that as meant to be solve so that it evaluates to zero. For our example, we rewrite

$$x = x^2$$

as

$$x - x^2 = 0$$

and then pass this to the solve function.

Let's repeat the same for the equation:

$$x = x^3$$

and solve

```
>>> eq = x - x ** 3                          # define the equation
>>> sympy.solve(eq, x)                       # solve eq = 0
[-1, 0, 1]
```

12.1.9 Output: LaTeX interface and pretty-printing

As is the case with many computer algebra systems, SymPy has the ability to format its output as LaTeX code, for easy inclusion into documents. By default, we get an inline equation, but we may request a displayed equation instead.

```
>>> series(1/(x+y), y, 0, 3)
1/x - y/x**2 + y**2/x**3 + O(y**3)
>>> print latex(series(1/(x+y), y, 0, 3))
$\frac{1}{x} - \frac{y}{x^{2}} + \frac{y^{2}}{x^{3}} + \operatorname{\
   mathcal{O}}\left(y^{3}\right)$
>>> print latex(series(1/(x+y), y, 0, 3), inline=False)
\begin{equation*}\frac{1}{x} - \frac{y}{x^{2}} + \frac{y^{2}}{x^{3}} +
    \operatorname{\mathcal{O}}\left(y^{3}\right)\end{equation*}
```

Note that the print command is important: omitting it will lead to Python escaping the backslashes in the above to indicate that they are literal, by adding further backslashes, which will affect the LaTeX rendering. Also be aware that in its default mode, `latex()` outputs code that requires the `amsmath` package to be loaded via a `\usepackage{amsmath}` command in the document preamble.

The above series is thus rendered:

$$\frac{1}{x} - \frac{y}{x^2} + \frac{y^2}{x^3} + \mathcal{O}\left(y^3\right)$$

SymPy also supports a "pretty print" (`pprint()`) output routine, which produces better-formatted ASCII output than the default printing routine:

```
>>> cos(x).series(x, 0.5, 10)
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)
>>> pprint(cos(x).series(x,0.5,10))
     2    4     6      8
    x    x     x      x
1 - -- + -- - --- + ----- + O(x**10)
    2    24   720   40320
>>> integrate(x**2*exp(y), x, x)
x**4*exp(y)/12
>>> pprint(integrate(x**2*exp(y), x, x))
 4  y
x .e
-----
  12
```

If your system is set up optimally, it is likely that `pprint()` output will look even better than the above example, as illustrated in figure 12.2. Note features such as the subscripts for array elements

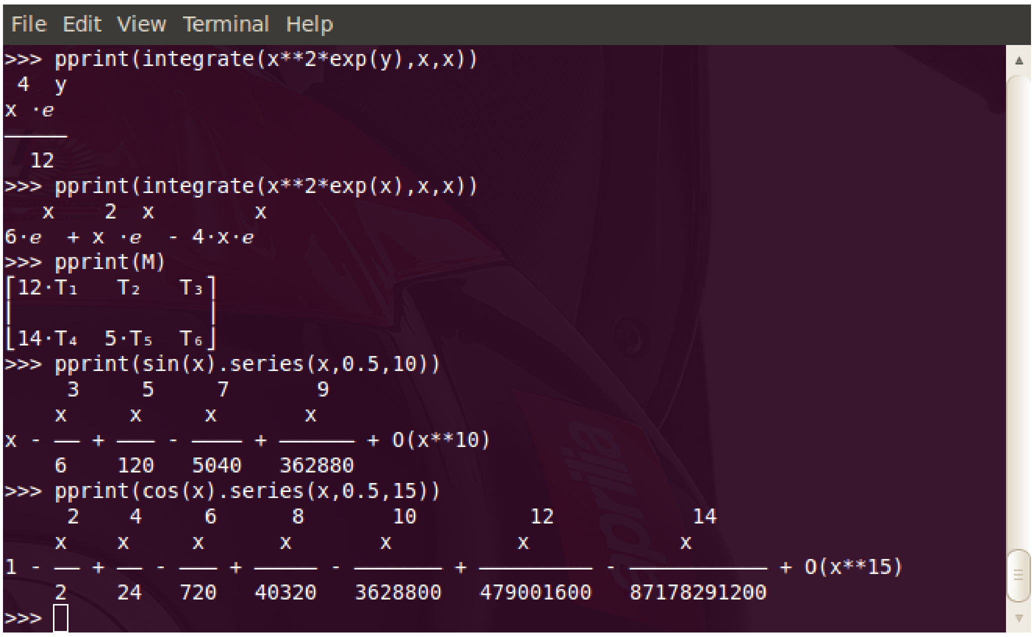


Figure 12.2: Nicely-formatted output from `pprint()`.

whose names are of the form `T_n`, the italicised constant e, vertically-centred dots for multiplication, and the nicely-formed matrix borders and fractions.

Finally, SymPy offers `preview()`, which displays rendered output on screen:

```
>>> preview(integrate(x**2*exp(y), x, x))
```

The last command results in a rendered formula being displayed, so long as appropriate LaTeX tools are available on the system[2]. It can display output in various formats (currently PNG, DVI, PostScript or PDF). See `help(preview)` for full details.

12.1.10 Automatic generation of C code

A strong point of many symbolic libraries is that they can convert the symbolic expressions to C-code (or other code) that can subsequently be compiled for high execution speed. Here is an example that demonstrates this:

```
>>> from sympy import *
>>> from sympy.utilities.codegen import codegen
>>> x = Symbol('x')
>>> sin(x).series(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> print codegen(("taylor_sine",sin(x).series(x,0,6)),
    "C", "myHeaderFile")[0][1]
/******************************************************
 *            Code generated with sympy 0.6.6         *
 *                                                    *
 *   See http://www.sympy.org/ for more information.*
 *                                                    *
 *              This file is part of 'project'       *
 ******************************************************/
```

[2] from the `help(preview)` documentation: "Currently this depends on pexpect, which is not available for windows."

```
#include "myHeaderFile.h"
#include <math.h>

double taylor_sine(double x) {
  return x - pow(x,3)/6 + pow(x,5)/120 + O(pow(x,6));
}
```

12.2 Related tools

It is worth noting that the SAGE initiative http://www.sagemath.org/ is trying to "create a viable free open source alternative to Magma, Maple, Mathematica and Matlab." and includes the SymPy library among many others. Its symbolic capabilities are more powerful than SymPy's, and SAGE, but the SymPy features will already cover many of the needs arising in science and engineering. SAGE includes the computer algebra system Maxima, which is also available standalone from http://maxima.sourceforge.net/.

Chapter 13

Numerical Computation

13.1 Numbers and numbers

We have already seen (3.2) that Python knows different *types* of numbers:

- `float`ing point numbers such as 3.14
- `int`egers such as 42
- `complex` numbers such as $3.14 + 1j$

There is one more type which is called `long`. This is a special integer type which can represent numbers up to any upper or lower limit (but computations with it are carried out much slower than using the `int` integer type).

13.1.1 Limitations of number types

Limitations of `ints`

Mathematics provides the infinite set of natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$. Because the computer has *finite* size, it is impossible to represent all of these numbers in the computer. Instead, only a small subset of numbers is represented.

The `int`-type can (usually[1]) represent numbers between -2147483648 and +2147483647 and corresponds to 4 bytes (that's 4*8 bit, and 2^{32} = 4294967296 which is the range from -2147483648 and +2147483647).

You can imagine that the hardware uses a table like this to encode integers using bits (suppose for simplicity we use only 8 bits for this):

natural number	bit-representation
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
⋮	⋮
254	11111110
255	11111111

[1]The exact value for the upper limit is availabe in `sys.maxint`.

Using 8 bit we can represent 256 natural numbers (for example from 0 to 255) because we have $2^8 = 256$ different ways of combining eight 0s and 1s.

We could also use a slightly different table to describe 256 integer numbers ranging, for example, from -127 to +128.

This is *in principle* how integers are represented in the computer. Depending on the number of bytes used, only integer numbers between a minimum and a maximum value can be represented. On today's hardware, it is common to use 4 or 8 bytes to represent one integer, which leads exactly to the minimum and maximum values of -2147483648 and +2147483647 as shown above for 4 bytes, and +9223372036854775807 as the maximum integer for 8 bytes (that's $\approx 9.2 \cdot 10^{18}$).

Limitations of `floats`

The floating point numbers in a computer are not the same as the mathematical floating point numbers. (This is exactly the same as the (mathematical) integer numbers not being the same as the integer numbers in a computer: only a *subset* of the infinite set of integer numbers can be represented by the `int` data type as shown in section 13.1). So how are floating point numbers represented in the computer?

- Any real number x can be written as

$$x = a \cdot 10^b$$

  where a is the mantissa and b the exponent.

- Examples:

x		a	b
123.456	$= 1.23456 \cdot 10^2$	1.23456	2
100000	$= 1.0 \cdot 10^6$	1.00000	6
0.0000024	$= 2.4 \cdot 10^{-6}$	2.40000	-6
Can write a and b as integers:			
		123456	2
		100000	6
		240000	-6

- Therefore, we can use 2 integers to encode one floating point number!

$$x = a \cdot 10^b$$

- Following (roughly) the IEEE-754 standard, one uses 8 bytes for one float x: these 64 bits are split as

  ▷ 10 bit for the exponent b and

  ▷ 54 bit for the mantissa a.

  This results in

  ▷ largest possible float $\approx 10^{308}$ (quality measure for b)

  ▷ smallest possible (positive) float $\approx 10^{-308}$ (quality measure for b)

  ▷ distance between 1.0 and next larger number $\approx 10^{-16}$ (quality measure for a)

Note that this is *in principle* how floating point numbers are stored (it is actually a bit more complicated).

Limitations of `complex` numbers

The `complex` number type has essentially the same limitations as the `float` data type (see section 13.1.1) because a complex number consists of two `floats`: one represents the real part, the other one the imaginary part.

. . . are these number types of practical value?

In practice, we do not usually find numbers in our daily life that exceed 10^{300} (this is a number with 300 zeros!), and therefore the floating point numbers cover the range of numbers we usually need.

However, be warned that in scientific computation small and large numbers are used which may (often in intermediate results) exceed the range of floating point numbers.

- Imagine for example, that we have to take the fourth power of the constant $\hbar = 1.0545716 \cdot 10^{-34}\text{kgm}^2/\text{s}$:
- $\hbar^4 = 1.2368136958909421 \cdot 10^{-136}\text{kg}^4\text{m}^8/\text{s}^4$ which is "halfway" to our representable smallest positive float of the order of 10^{-308}.

If there is any danger that we might exceed the range of the floating point numbers, we have to *rescale* our equations so that (ideally) all numbers are of order unity. Rescaling our equations so that all relevant numbers are approximately 1 is also useful in debugging our code: if numbers much greater or smaller than 1 appear, this may be an indication of an error.

13.1.2 Using floating point numbers (carelessly)

We know already that we need to take care that our floating point values do not exceed the range of floating point numbers that can be represented in the computer (see 13.1.1).

There is another complication due to the way floating point numbers have to be represented internally: not all floating point numbers can be represented exactly in the computer. The number 1.0 can be represented exactly but the numbers 0.1, 0.2 and 0.3 cannot:

```
>>> 1.0
1.0
>>> 0.1
0.10000000000000001
>>> 0.2
0.20000000000000001
>>> 0.3
0.29999999999999999
```

Instead, the floating point number "nearest" to the real number is chosen. For 0.1 this is 0.10000000000000001, and for 0.3 this is 0.29999999999999999.

This can cause problems. Suppose we need a loop where x takes values 0.1, 0.2, 0.3, . . . , 0.9, 1.0. We might be tempted to write something like this:

```
x = 0.0
while not x == 1.0:
    x = x + 0.1
    print("x=%19.17g" % (x))
```

However, this loop will never terminate. Here are the first 11 lines of output of the program:

```
x=0.10000000000000001
x=0.20000000000000001
x=0.30000000000000004
x=0.40000000000000002
x=                  0.5
x=0.59999999999999998
x=0.69999999999999996
x=0.79999999999999993
x=0.89999999999999991
x=0.99999999999999989
x=1.0999999999999999
```

Because the variable `x` never takes exactly the value 1.0, the while loop will continue forever.

Thus: *Never compare two floating point numbers for equality.*

13.1.3 Using floating point numbers carefully 1

There are a number of alternative ways to solve this problem. For example, we can compare the distance between two floating point numbers:

```
x = 0.0
while abs(x - 1.0) > 1e-8:
    x = x + 0.1
    print("x=%19.17g" % (x))
```

13.1.4 Using floating point numbers carefully 2

Alternatively, we can (for this example) iterate over a sequence of integers and compute the floating point number from the integer:

```
for i in range(1, 11):
    x = i * 0.1
    print("x=%19.17g" % (x))
```

It is worth showing the output of this program:

```
x=0.10000000000000001
x=0.20000000000000001
x=0.30000000000000004
x=0.40000000000000002
x=                  0.5
x=0.60000000000000009
x=0.70000000000000007
x=0.80000000000000004
x=0.90000000000000002
x=                    1
```

If we compare this with the output from the program in section 13.1.2, we can see that the floating point numbers differ. This means that – in a numerical calculation – it is not true that $0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 1.0$.

13.1.5 Symbolic calculation

Using the sympy package we have arbitrary precision. Using `sympy.Rational`, we can define the fraction 1/10 exactly symbolically. Adding this 10 times will lead exactly to the value 1, as demonstrated by this script

```
from sympy import Rational
dx = Rational(1,10)
x = 0
while x != 1.0:
    x = x+dx
    print "Current x=%4s = %3.1f" % (x,x.evalf())
print "Reached x=%s" % x
```

and its output

```
Current x=1/10 = 0.1
Current x= 1/5 = 0.2
Current x=3/10 = 0.3
Current x= 2/5 = 0.4
Current x= 1/2 = 0.5
Current x= 3/5 = 0.6
Current x=7/10 = 0.7
Current x= 4/5 = 0.8
Current x=9/10 = 0.9
Current x=   1 = 1.0
Reached x=1
```

However, this symbolic calculation is much slower as it is done through software rather than the CPU-based floating point operations. The next program approximates the relative performances:

```
from sympy import Rational
dx_symbolic = Rational(1,10)
dx = 0.1

def loop_sympy(n):
    x=0
    for i in xrange(n):
        x = x + dx_symbolic
    return x

def loop_float(n):
    x=0
    for i in xrange(n):
        x = x + dx
    return x

def time_this(f,n):
    import time
    starttime = time.time()
    result = f(n)
    stoptime = time.time()
    print "deviation is %16.15g" % (n*dx_symbolic-result)
```

```
    return stoptime-starttime

n=100000
print "loop using float dx:"
time_float = time_this(loop_float,n)
print "float loop n=%d takes %6.5f seconds" % (n,time_float)
print "loop using sympy symbolic dx:"
time_sympy = time_this(loop_sympy,n)
print "sympy loop n=%d takes %6.5f seconds" % (n,time_sympy)
print "Symbolic loop is a factor %.1f slower." % (time_sympy/time_float)
```

Output (run on MacBook Pro September 2011):

```
loop using float dx:
deviation is -1.88483681995422e-08
float loop n=100000 takes 0.01390 seconds
loop using sympy symbolic dx:
deviation is                 0
sympy loop n=100000 takes 2.75708 seconds
Symbolic loop is a factor 198.4 slower.
```

This is of course an artificial example: we have added the symbolic code to demonstrate that these round off errors originate from the approximative representation of floating point numbers in the hardware (and thus programming languages). We can, in principle, avoid these complications by computing using symbolic expressions, but this is in practice too slow.[2]

13.1.6 Summary

In summary, we have learned that

- floating point numbers and integers used in numeric computation are generally quite different from "mathematical numbers" (symbolic calculations are exact and use the "mathematical numbers"):
    - there is a maximum number and a minimum number that can be represented (for both integers and floating point numbers)
    - within this range, not every floating point number can be represented in the computer.
- We deal with this limitation by:
    - never comparing two floating point numbers for equality (instead we compute the absolute value of the difference)
    - use of algorithms that are *stable* (this means that small deviations from correct numbers can be corrected by the algorithm. We have not yet shown any such examples this document.)
- Note that there is a lot more to be said about numerical and algorithmic tricks and methods to make numeric computation as accurate as possible but this is outside the scope of this section.

[2]We add for completeness, that a C-program (or C++ of Fortran) that executes the same loop will be about 100 times faster than the python float loop, and thus about 100*200 = 20000 faster than the symbolic loop.

13.1.7 Exercise: infinite or finite loop

1. What does the following piece of code compute? Will the loop ever finish? Why?

```
eps = 1.0
while 1.0 + eps > 1.0:
    eps = eps / 2.0
print(eps)
```

Chapter 14

Numerical Python (numpy): arrays

14.1 Numpy introduction

The NumPy package (read as NUMerical PYthon) provides access to

- a new data structure called `array`s which allow
- efficient vector and matrix operations. It also provides
- a number of linear algebra operations (such as solving of systems of linear equations, computation of Eigenvectors and Eigenvalues).

14.1.1 History

Some background information: There are two other implementations that provide nearly the same functionality as NumPy. These are called "Numeric" and "numarray":

- Numeric was the first provision of a set of numerical methods (similar to Matlab) for Python. It evolved from a PhD project.
- Numarray is a re-implementation of Numeric with certain improvements (but for our purposes both Numeric and Numarray behave virtually identical).
- Early in 2006 it was decided to merge the best aspects of Numeric and Numarray into the Scientific Python (`scipy`) package and to provide (a hopefully "final") `array` data type under the module name "NumPy".

We will use in the following materials the "NumPy" package as provided by (new) SciPy. If for some reason this doesn't work for you, chances are that your SciPy is too old. In that case, you will find that either "Numeric" or "numarray" is installed and should provide nearly the same capabilities.[1]

14.1.2 Arrays

We introduce a new data type (provided by NumPy) which is called "`array`". An array *appears* to be very similar to a list but an array can keep only elements of the same type (whereas a list can mix different kinds of objects). This means arrays are more efficient to store (because we don't need to store the type for every element). It also makes arrays the data structure of choice for numerical calculations where we often deal with vectors and matricies.

Vectors and matrices (and matrices with more than two indices) are all called "arrays" in NumPy.

[1] In this text, we usually import `numpy` under the name `N` like this: `import numpy as N`. If you don't have `numpy` on your machine, you can substitute this line by `import Numeric as N` or `import numarray as N`.

Vectors (1d-arrays)

The data structure we will need most often is a vector. Here are a few examples of how we can generate one:

- Conversion of a list (or tuple) into an array using `numpy.array`:

```
>>> import numpy as N
>>> x = N.array([0, 0.5, 1, 1.5])
>>> print(x)
[ 0.   0.5  1.   1.5]
```

- Creation of a vector using "ArrayRANGE":

```
>>> x = N.arange(0, 2, 0.5)
>>> print(x)
[ 0.   0.5  1.   1.5]
```

- Creation of vector with zeros

```
>>> x = N.zeros(4)
>>> print(x)
[ 0.  0.  0.  0.]
```

Once the array is established, we can set and retrieve individual values. For example:

```
>>> x = N.zeros(4)
>>> x[0] = 3.4
>>> x[2] = 4
>>> print(x)
[ 3.4  0.   4.   0. ]
>>> print(x[0])
3.4
>>> print(x[0:-1])
[ 3.4  0.   4. ]
```

Note that once we have a vector we can perform calculations on every element in the vector with a single statement:

```
>>> x = N.arange(0, 2, 0.5)
>>> print(x)
[ 0. ,  0.5,  1. ,  1.5]
>>> print(x + 10)
[ 10.   10.5  11.   11.5]
>>> print(x ** 2)
[ 0.    0.25  1.    2.25]
>>> print(N.sin(x))
[ 0.          0.47942554  0.84147098  0.99749499]
```

Matrices (2d-arrays)

Here are two ways to create a 2d-array:

- By converting a list of lists (or tuples) into an array:

```
>>> x = N.array([[1, 2, 3], [4, 5, 6]])
>>> x
[[1 2 3]
 [4 5 6]]
```

- Using the `zeros` method to create a matrix with 5 rows and 4 columns

```
>>> x = N.zeros((5, 4))
>>> print x
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

The "shape" of a matrix can be queried like this (here we have 2 rows and 3 columns):

```
>>> x=N.array([[1, 2, 3], [4, 5, 6]])
>>> print x
[[1 2 3]
 [4 5 6]]
>>> x.shape
(2, 3)
```

Individual elements can be accessed and set using this syntax:

```
>>> x=N.array([[1, 2, 3], [4, 5, 6]])
>>> x[0, 0]
1
>>> x[0, 1]
2
>>> x[0, 2]
3
>>> x[1, 0]
4
>>> x[:, 0]
array([1, 4])
>>> x[0,:]
array([1, 2, 3])
```

14.1.3 Convert from array to list or tuple

To create an array back to a list or tuple, we can use the standard python functions `list(s)` and `tuple(s)` which take a sequence `s` as the input argument and return a list and tuple, respectively:

```
>>> a = N.array([1, 4, 10])
>>> a
array([1,  4, 10])
>>> list(a)
[1, 4, 10]
```

```
>>> tuple(a)
(1, 4, 10)
```

14.1.4 Standard Linear Algebra operations

Maxtrix multiplication

Two arrays can be multiplied in the usual linear-algebra way using `numpy.matrixmultiply`. Here is an example:

```
>>> import numpy as N
>>> import numpy.random
>>> A = numpy.random.rand(5, 5)    # generates a random 5 by 5 matrix
>>> x = numpy.random.rand(5)       # generates a 5-element vector
>>> b=N.dot(A, x)                  # multiply matrix A with vector x
```

Solving systems of linear equations

To solve a system of equations $A\mathbf{x} = \mathbf{b}$ that is given in matrix form (*i.e* A is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors where A and $\mathbf{b}$ are known and we want to find the unknown vector $\mathbf{x}$), we can use the linear algebra package (`linalg`) of `numpy`:

```
>>> import numpy.linalg as LA
>>> x = LA.solve(A, b)
```

Computing Eigenvectors and Eigenvalues

Here is a small example that computes the [trivial] Eigenvectors and Eigenvalues (`eig`) of the unity matrix (`eye`)):

```
>>> import numpy
>>> import numpy.linalg as LA
>>> A = numpy.eye(3)      #'eye'->I->1 (ones on the diagonal)
>>> print A
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>> evalues, evectors = LA.eig(A)
>>> print(evalues)
[ 1.+0.j  1.+0.j  1.+0.j]
>>> print(evectors)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Note that each of these commands provides its own documentation. For example, `help(LA.eig)` will tell you all about the eigenvector and eigenvalue function (once you have imported `numpy.linalg` as `LA`).

Curve fitting of polynomials

Let's assume we have x-y data to which we like to fit a curve (to minimise the least square deviation of the fit from the data).

Numpy provides the routine `polyfit(x,y,n)` (which is similar to Matlab's `polyfit` function which takes a list `x` of x-values for data points, a list `y` of y-values of the same data points and a desired order of the polynomial that will be determined to fit the data in the least-square sense as well as possible.

```
import numpy

#demo curve fitting: xdata and ydata are input data
xdata = numpy.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
ydata = numpy.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])

#now do fit for cubic (order = 3) polynomial
z = numpy.polyfit(xdata, ydata, 3)

#z is an array of coefficients, highest first, i.e.
#          x^3          X^2            X            0
#z=array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254])

#It is convenient to use `poly1d` objects for dealing with polynomials:
p = numpy.poly1d(z)   # creates a polynomial function p from coefficients
                      # and p can be evaluated for all x then.

#create plot
xs = [0.1 * i for i in range(50)]
ys = [p(x) for x in xs]      # evaluate p(x) for all x in list xs

import pylab
pylab.plot(xdata, ydata, 'o', label='data')
pylab.plot(xs, ys, label='fitted curve')
pylab.ylabel('y')
pylab.xlabel('x')
pylab.savefig('polyfit.pdf')
pylab.show()
```

Figure 14.1 shows the fitted curve (solid line) together with the precise computed data points.

14.1.5 More numpy examples...

...can be found here: http://www.scipy.org/Numpy_Example_List

14.1.6 Numpy for Matlab users

There is a dedicated webpage that explains Numpy from the perspective of a (experienced) Matlab user at `http://www.scipy.org/NumPy_for_Matlab_Users`.

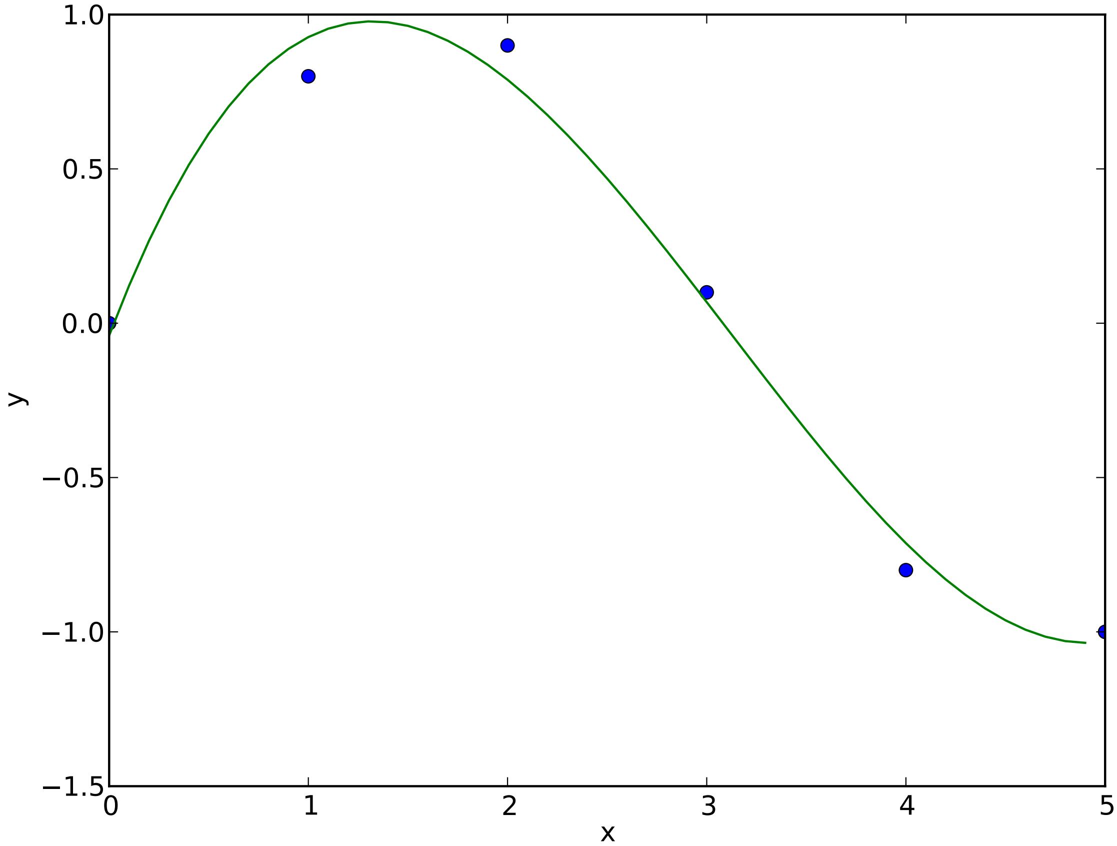


Figure 14.1: Demonstration of least-squares curvefitting with numpy

Chapter 15

Visualising Data

The purpose of scientific computation is insight not numbers: To understand the meaning of the (many) numbers we compute, we often need postprocessing, statistical analysis and graphical visualisation of our data. The following sections describe

- Matplotlib/Pylab — which allows us to generate high quality graphs of the type $y = f(x)$ (and a bit more)
- Visual Python — which is a very handy tool to quickly generate animations of time dependent processes taking place in 3d space.

15.1 Matplotlib (Pylab) – plotting y=f(x), (and a bit more)

The Python library *Matplotlib* is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments. Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code.

For more detailed information, check these links

- A very nice introduction in the object oriented Matplotlib interface, and summary of all important ways of changing style, figure size, linewidth, etc. This is a useful reference: http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-4-Matplotlib.ipynb
- Matplotlib tutorial http://matplotlib.sourceforge.net/users/index.html
- Matplotlib home page http://matplotlib.sourceforge.net
- List of simple screenshot examples http://matplotlib.sourceforge.net/users/screenshots.html
- Extended thumbnail gallery of examples http://matplotlib.sourceforge.net/gallery.html

15.1.1 Matplotlib and Pylab

Matplotlib as *an object oriented plotting library.* Pylab is an interface to the same set of functions that imitates the (state-driven) Matlab plotting interface.

Pylab is slightly more convenient to use for easy plots, and Matplotlib gives far more detailed control over how plots are created. If you use Matplotlib routinely to produce figures, you are well advised to learn about the object oriented matplotlib interface (instead of the pylab interface).

This chapter focusses on the Pylab interface.

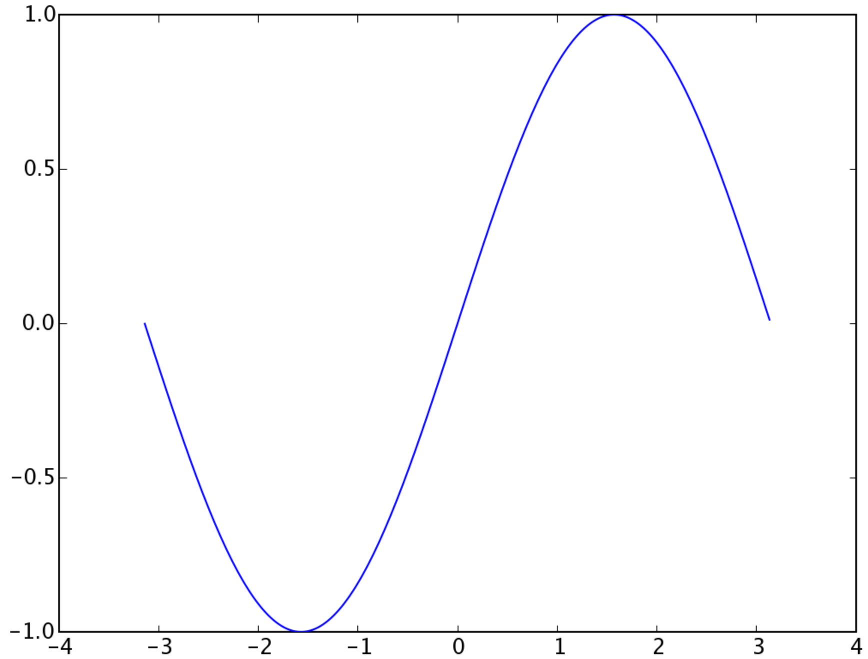


Figure 15.1: The output of `pylab1.py`.

An excellent introduction and overview of the Matplotlib plotting interface is available in http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-4-Matplotlib.ipynb.

15.1.2 First example

The recommended way of using Matplotlib in a simple example is shown here (let's call this example 1a, the output is shown in figure 15.1):

```
#example 1a
import numpy as np                      # get access to fast arrays
import matplotlib.pyplot as plt         # the plotting functions

x = np.arange(-3.14, 3.14, 0.01)        # create x-data
y = np.sin(x)                           # compute y-data
plt.plot(x, y)                          # create plot
plt.show()                              # show plot (makes window pop up)
```

15.1.3 How to import matplotlib, pylab, pyplot, numpy and all that

The submodule `matplotlib.pyplot` provides an object oriented interface to the plotting library. Many of the examples in the matplotlib documentation follow the import convention to import `matplotlib.pyplot` as `plt` and `numpy` as `np`. It is of course entirely the user's decision whether to import the `numpy` library under the name `np` (as often done in matplotlib examples) or `N` as done in this text (and in the early days when the predecessor of numpy was called "Numeric") or any other name you like. Similarly, it is a matter of taste whether the plotting submodule (`matplotlib.pyplot`) is

imported as plt as is done in the matplotlib documentation or plot (which could be argued is slightly clearer) etc.

As always a balance has to be struck between personal preferences and consistency with common practice in choosing these name. Consistency with common use is of course more important if the code is likely to be used by others or published.

Plotting nearly always needs arrays of numerical data and it is for this reason that the numpy module is used a lot: it provides fast and memory efficient array handling for Python (see section 14.1).

We could thus also have written the example 1a above as in example 1b (which is identical in functionality to the example above and will create the plot shown in figure 15.1):

```
import pylab
import numpy as N

x = N.arange(-3.14, 3.14, 0.01)
y = N.sin(x)

pylab.plot(x, y)
pylab.show()
```

Because the numpy.arange and numpy.sin objects have already been imported into the (convenience) pylab namespace, we could also write it as example 1c:

```
"""example 1c"""
import pylab as p

x = p.arange(-3.14, 3.14, 0.01)
y = p.sin(x)

p.plot(x, y)
p.show()
```

If we really want to cut down on characters to type, we could also important the whole functionality from the pylab convenience module, and rewrite the code as example 1d:

```
"""example 1d"""
from pylab import *              # not generally recommended
                                 # okay for interactive testing

x = arange(-3.14, 3.14, 0.01)
y = sin(x)

plot(x, y)
show()
```

This can be extremely convenient, but comes with a big health warning:

- While using `from pylab import *` is acceptable at the command prompt to interactively create plots and analyse data, this should never be used in any plotting scripts.

- The pylab toplevel provides over 800 different objects which are all imported into the global name space when running `from pylab import *`. This is not good practice, and could conflict with other objects that exist already or are created later.

- As a rule of thumb: do never use `from somewhere import *` in programs we save. This may be okay at the command prompt.

In the following examples, we usually use the `pylab` interface to the plotting routines but this is purely a matter of taste and habit and by no means the only way (note that the Matplotlib authors recommend the import style as in example 1a, see also this Matplot FAQ entry: Matplotlib, pylab, and pyplot: how are they related?)

15.1.4 IPython's inline mode

If you have the IPython qtconsole or notebook installed (see section 11.3.1), then we can use the `%matplotlib inline` magic command to make further plots appear within our console or notebook. To force pop up windows instead, use `%matplotlib qt`.

There is also the `%pylab` magic, which will not only switch to inline plotting but also automatically execute `from pylab import *`.

15.1.5 Saving the figure to a file

Once you have created the figure (using the `plot` command) and added any labels, legends etc, you have two options to save the plot.

1. You can display the figure (using `show`) and *interactively* save it by clicking on the disk icon.
2. You can (without displaying the figure) save it directly from your Python code. The command to use is `savefig`. The format is determined by the extension of the file name you provide. Here is an example (`pylabsavefig.py`) which saves the plot shown in Figure 15.1.

```
import pylab
import numpy as N

x = N.arange(-3.14, 3.14, 0.01)
y = N.sin(x)

pylab.plot(x, y, label='sin(x)')
pylab.savefig('myplot.png')     # saves png file
pylab.savefig('myplot.eps')     # saves eps file
pylab.savefig('myplot.pdf')     # saves pdf file
```

A note on file formats: Choose the `png` file format if you plan to include your graph in a word document or on a webpage. Choose the `eps` or `pdf` file format if you plan to include the figure in a LaTeXdocument – depending on whether you want to compile it using '`latex` (needs `eps`) or pdflatex (can use `pdf` [better] or `png`). If the version of MS Word (or other text processing software you use) can handle `pdf` files, it is better to use `pdf` than `png`.

Both `pdf` and `eps` are vector file formats which means that one can zoom into the image without loosing quality (lines will still be sharp). File formats such as `png` (and `jpg`, `gif`, `tif`, `bmp`) save the image in form of a bitmap (i.e. a matrix of colour values) and will appear blurry or pixelated when zooming in (or when printed in high resolution).

15.1.6 Interactive mode

Matplotlib can be run in two modes:

- non-interactive (this is the default)

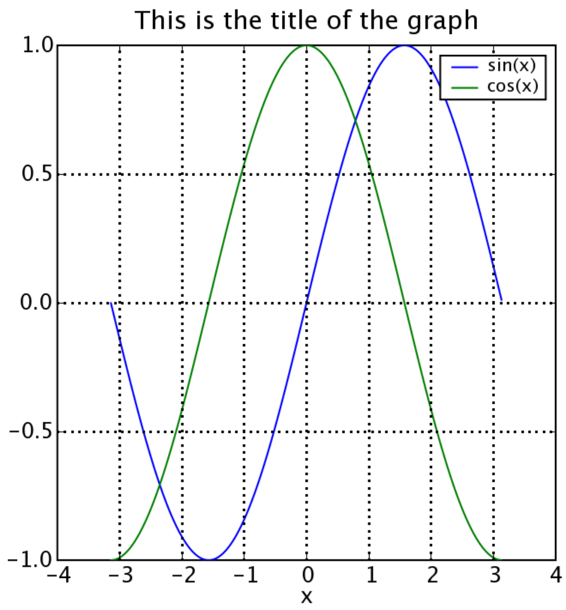


Figure 15.2: The output of `pylab2.py`.

- interactive.

In non-interactive mode, no plots will be displayed until the `show()` command has been issued. In this mode, the `show()` command should be the last statement of your program.

In interactive mode, plots will be immediately displayed after the plot command has been issued. One can switch the interactive mode on using `pylab.ion()` and off using `pylab.ioff()`.

15.1.7 Fine tuning your plot

Matplotlib allows us to fine tune our plots in great detail. Here is an example (`pylab2.py`) producing figure 15.2:

```
import pylab
import numpy as N

x = N.arange(-3.14, 3.14, 0.01)
y1 = N.sin(x)
y2 = N.cos(x)

pylab.figure(figsize=(5, 5))
pylab.plot(x, y1, label='sin(x)')
pylab.plot(x, y2, label='cos(x)')
pylab.legend()
pylab.grid()
pylab.xlabel('x')
pylab.title('This is the title of the graph')
pylab.show()              # necessary to display graph
```

showing some other useful commands:

- `figure(figsize=(5,5))` sets the figure size to 5inch by 5inch

- `plot(x,y1,label='sin(x)')` The "label" keyword defines the name of this line. The line label will be shown in the legend if the `legend()` command is used later.
- Note that calling the `plot` command repeatedly, allows you to overlay a number of curves.
- `axis([-2,2,-1,1])` This fixes the displayed area to go from xmin=-2 to xmax=2 in x-direction, and from ymin=-1 to ymax=1 in y-direction
- `legend()` This command will display a legend with the labels as defined in the plot command. Try `help("pylab.legend")` to learn more about the placement of the legend.
- `grid()` This command will display a grid on the backdrop.
- `xlabel('...')` and `ylabel('...')` allow labelling the axes.

Note further than you can chose different line styles, line thicknesses, symbols and colours for the data to be plotted. (The syntax is very similar to MATLAB.) For example:

- `plot(x,y,'og')` will plot circles (`o`) in green (`g`)
- `plot(x,y,'-r')` will plot a line (`-`) in red (`r`)
- `plot(x,y,'-b',linewidth=2)` will plot a blue line (`b`) with two two pixel thickness `linewidth=2` which is twice as wide as the default.

The full list of options can be found when typing `help("pylab.plot")` at the Python prompt. Because this documentation is so useful, we repeat parts of it here:

```
plot(*args, **kwargs)
    Plot lines and/or markers to the
    :class:`~matplotlib.axes.Axes`.  *args* is a variable length
    argument, allowing for multiple *x*, *y* pairs with an
    optional format string.  For example, each of the following is
    legal::

        plot(x, y)          # plot x and y using default line style and color
        plot(x, y, 'bo')    # plot x and y using blue circle markers
        plot(y)             # plot y using x as index array 0..N-1
        plot(y, 'r+')       # ditto, but with red plusses

    If *x* and/or *y* is 2-dimensional, then the corresponding columns
    will be plotted.

    An arbitrary number of *x*, *y*, *fmt* groups can be
    specified, as in::

        a.plot(x1, y1, 'g^', x2, y2, 'g-')

    Return value is a list of lines that were added.

    The following format string characters are accepted to control
    the line style or marker:

    ================    ===============================
    character           description
    ================    ===============================
    '-'                 solid line style
    '--'                dashed line style
    '-.'                dash-dot line style
    ':'                 dotted line style
```

```
    '.'                 point marker
    ','                 pixel marker
    'o'                 circle marker
    'v'                 triangle_down marker
    '^'                 triangle_up marker
    '<'                 triangle_left marker
    '>'                 triangle_right marker
    '1'                 tri_down marker
    '2'                 tri_up marker
    '3'                 tri_left marker
    '4'                 tri_right marker
    's'                 square marker
    'p'                 pentagon marker
    '*'                 star marker
    'h'                 hexagon1 marker
    'H'                 hexagon2 marker
    '+'                 plus marker
    'x'                 x marker
    'D'                 diamond marker
    'd'                 thin_diamond marker
    '|'                 vline marker
    '_'                 hline marker
    ================    ===============================


    The following color abbreviations are supported:

    ==========  ========
    character   color
    ==========  ========
    'b'         blue
    'g'         green
    'r'         red
    'c'         cyan
    'm'         magenta
    'y'         yellow
    'k'         black
    'w'         white
    ==========  ========

    In addition, you can specify colors in many weird and
    wonderful ways, including full names (``'green'``), hex
    strings (``'#008000'``), RGB or RGBA tuples (``(0,1,0,1)``) or
    grayscale intensities as a string (``'0.8'``).  Of these, the
    string specifications can be used in place of a ``fmt`` group,
    but the tuple forms can be used only as ``kwargs``.

    Line styles and colors are combined in a single format string, as in
    ``'bo'`` for blue circles.

    The *kwargs* can be used to set line properties (any property that has
    a ``set_*`` method).  You can use this to set a line label (for auto
    legends), linewidth, anitialising, marker face color, etc.  Here is an
    example::

        plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
        plot([1,2,3], [1,4,9], 'rs',  label='line 2')
        axis([0, 4, 0, 10])
        legend()

    If you make multiple lines with one plot command, the kwargs
    apply to all those lines, e.g.::
```

```
        plot(x1, y1, x2, y2, antialised=False)

    Neither line will be antialiased.

    You do not need to use format strings, which are just
    abbreviations.  All of the line properties can be controlled
    by keyword arguments.  For example, you can set the color,
    marker, linestyle, and markercolor with::

        plot(x, y, color='green', linestyle='dashed', marker='o',
             markerfacecolor='blue', markersize=12).  See
             :class:`~matplotlib.lines.Line2D` for details.
```

The use of different line styles and thicknesses is particularly useful when colour cannot be used to distinguish lines (for example when graph will be used in document that is to be printed in black and white only).

15.1.8 Plotting more than one curve

There are three different methods to display more than one curve.

Two (or more) curves in one graph

By calling the `plot` command repeatedly, more than one curve can be drawn in the same graph. Example:

```
import numpy as N
t=N.arange(0,2*N.pi,0.01)

import pylab
pylab.plot(t,N.sin(t),label='sin(t)')
pylab.plot(t,N.cos(t),label='cos(t)')
pylab.legend()
pylab.show()
```

Two (or more graphs) in one figure window

The `pylab.subplot` command allows to arrange several graphs within one figure window. The general syntax is

```
subplot(numRows, numCols, plotNum)
```

For example, to arrange 4 graphs in a 2-by-2 matrix, and to select the first graph for the next plot command, one can use:

```
subplot(2, 2, 1)
```

Here is a complete example plotting the sine and cosine curves in two graphs that are aligned underneath each other within the same window as shown in figure 15.3 (`pylabsubplots.py`):

```
import numpy as N
t = N.arange(0, 2 * N.pi, 0.01)

import pylab
pylab.subplot(2, 1, 1)
```

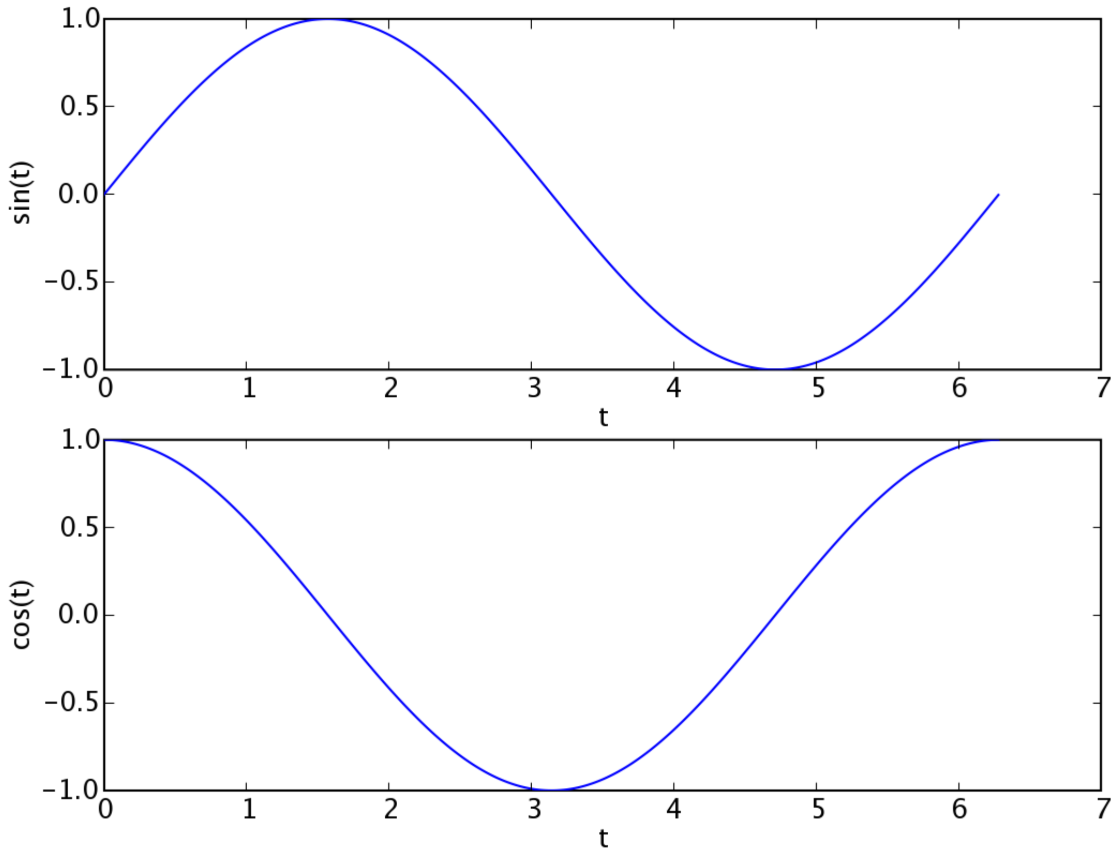


Figure 15.3: The output of `pylabsubplots.py`.

```
pylab.plot(t, N.sin(t))
pylab.xlabel('t')
pylab.ylabel('sin(t)')

pylab.subplot(2, 1, 2)
pylab.plot(t, N.cos(t))
pylab.xlabel('t')
pylab.ylabel('cos(t)')

pylab.show()
```

Two (or more) figure windows

```
import pylab
pylab.figure(1)
pylab.plot(range(10),'o')

pylab.figure(2)
pylab.plot(range(100),'x')
```

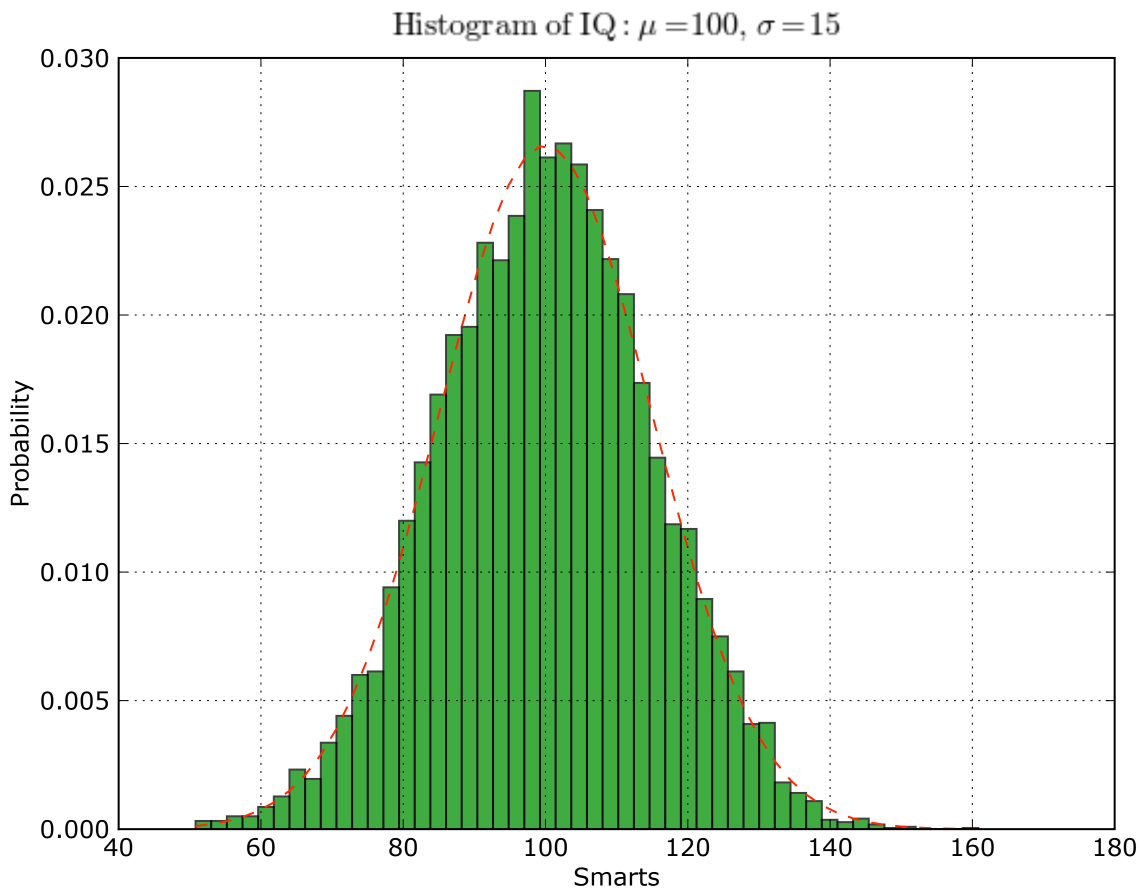


Figure 15.4: A histogram with fitted curve created with Matplotlib.

```
pylab.show()
```

Note that you can use `pylab.close()` to close one, some or all figure windows (use `help("pylab.close")` to learn more).

15.1.9 Histograms

The program below demonstrates how to create histograms from stastical data in Matplotlib. The resulting plot is show in figure

```
#modified version of
#http://matplotlib.sourceforge.net/plot_directive/mpl_examples/...
#                                  /pylab_examples/histogram_demo.py
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

# create data
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

```

```
# histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1,
                            facecolor='green', alpha=0.75)

#some finetuning of plot
plt.xlabel('Smarts')
plt.ylabel('Probability')
#Can use Latex strings for labels and titles:
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)


# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)

#save to file
plt.savefig('pylabhistogram.pdf')
#then display
plt.show()
```

Do not try to understand every single command in this file: some are rather specialised and have not been covered in this text. The intention is to provide a few examples to show what can – in principle – be done with Matplotlib. If you need a plot like this, the expectation is that you will need to experiment and possibly learn a bit more about Matplotlib.

15.1.10 Visualising matrix data

The program below demonstrates how to create a bitmap-plot of the entries of a matrix. The resulting plot is show in figure 15.5 and 15.6:

```
import numpy as np
import matplotlib.mlab as mlab     # Matlab compatibility commands
import matplotlib.pyplot as plt

#create matrix Z that contains some interesting data
delta = 0.1
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z = mlab.bivariate_normal(X, Y, 3.0, 1.0, 0.0, 0.0)

#display the 'raw' matrix data of Z in one figure
plt.figure(1)
plt.imshow(Z, interpolation='nearest')
plt.title("imshow example 1a: no interpolation")
plt.savefig("pylabimshow1a.pdf")

#display the data interpolated in other figure
plt.figure(2)
im = plt.imshow(Z, interpolation='bilinear')
```

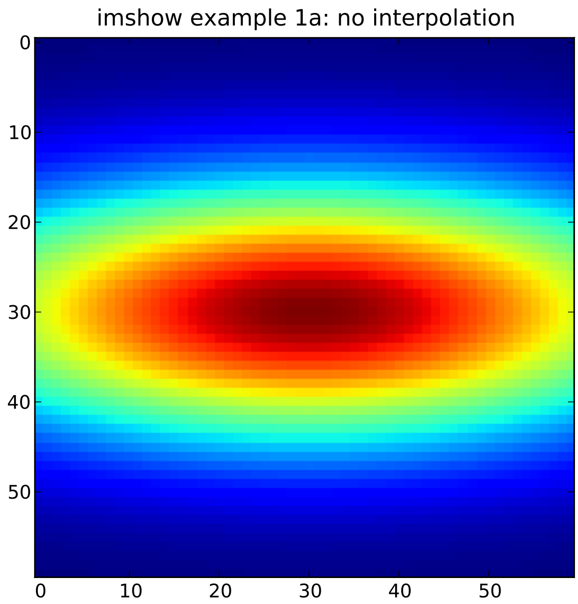


Figure 15.5: The assymmetric Gaussian stored in a matrix, visualised with the `imshow()` command *without* interpolation between matrix elements.

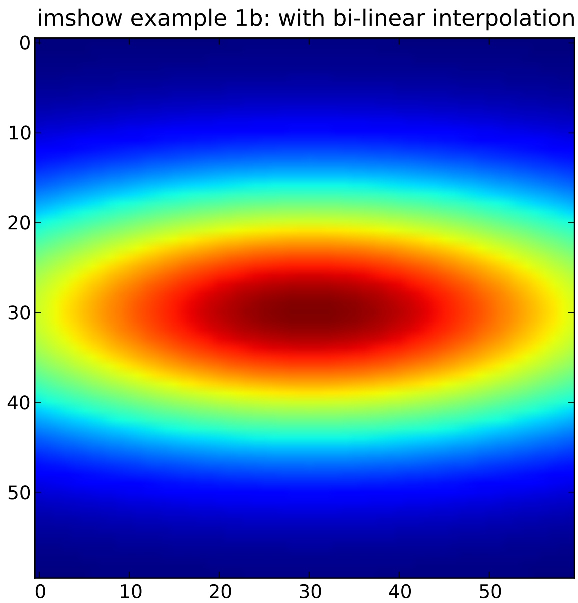


Figure 15.6: The assymmetric Gaussian stored in a matrix, visualised with the `imshow()` command with (bilinear) interpolation between matrix elements.

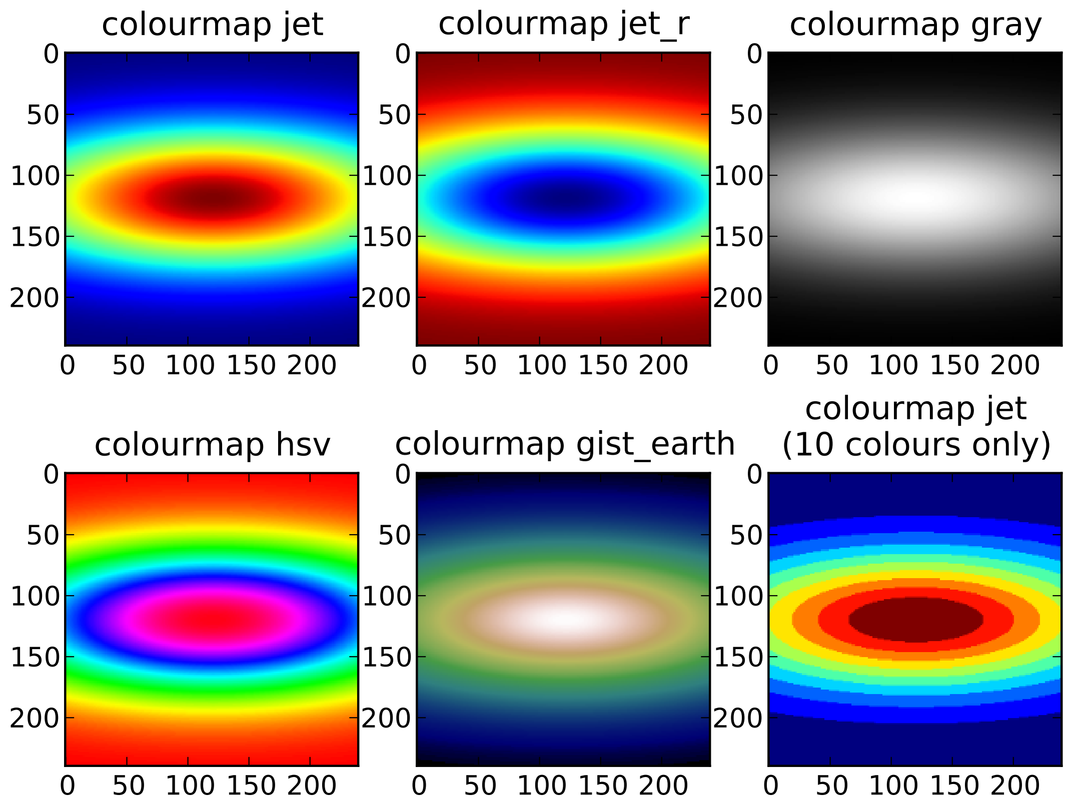


Figure 15.7: The assymmetric Gaussian stored in a matrix, visualised with the `imshow()` command using different colourmaps.

```
plt.title("imshow example 1b: with bi-linear interpolation")
plt.savefig("pylabimshow1b.pdf")

plt.show()
```

To use different colourmaps, we make use of the `matplotlib.cm` module (where cm stands for Colour Map). The code below demonstrates how we can select colourmaps from the set of already provided maps, and how we can modify them (here by reducing the number of colours in the map). The resulting plots are shown in figure 15.7. The last example mimics the behaviour of the more sophisticated `contour` command that also comes with `matplotlib`.

```
import numpy as np
import matplotlib.mlab as mlab     # Matlab compatibility commands
import matplotlib.pyplot as plt
import matplotlib.cm as cm         # Colour map submodule

#create matrix Z that contains some data interesting data
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
```

```
Z = mlab.bivariate_normal(X, Y, 3.0, 1.0, 0.0, 0.0)

Nx, Ny = 2, 3
plt.subplot(Nx, Ny, 1)  # next plot will be shown in
                        # first subplot in Nx x Ny
                        # matrix of subplots

plt.imshow(Z, cmap=cm.jet)  # default colourmap 'jet'
plt.title("colourmap jet")

plt.subplot(Nx, Ny, 2)   # next plot for second subplot
plt.imshow(Z, cmap=cm.jet_r)  # reverse colours in jet
plt.title("colourmap jet_r")

plt.subplot(Nx, Ny, 3)
plt.imshow(Z, cmap=cm.gray)
plt.title("colourmap gray")

plt.subplot(Nx, Ny, 4)
plt.imshow(Z, cmap=cm.hsv)
plt.title("colourmap hsv")

plt.subplot(Nx, Ny, 5)
plt.imshow(Z, cmap=cm.gist_earth)
plt.title("colourmap gist_earth")

plt.subplot(Nx, Ny, 6)
#make isolines by reducing number of colours to 10
mycmap = cm.get_cmap('jet', 10)    # 10 discrete colors
plt.imshow(Z, cmap=mycmap)
plt.title("colourmap jet\n(10 colours only)")

plt.savefig("pylabimshowcm.pdf")
plt.show()
```

15.1.11 Plots of $z = f(x, y)$ and other features of Matplotlib

Matplotlib has a large number of features and can create all the standard (1d and 2d) plots such as histograms, pie charts, scatter plots, 2d-intensity plots (i.e. $z = f(x, y)$) and contour lines) and much more. Figure 15.8 shows such an example (the `contour_demo.py` program is a standard example of the `pylab` package). This link provides source code to produce this kind of plot: contour_demo.py

Other examples are

- http://matplotlib.sourceforge.net/users/screenshots.html
- http://matplotlib.sourceforge.net/gallery.html
- Recently, creation of 3d-plots has been added to pylab: http://matplotlib.sourceforge.net/examples/mplot
  examples

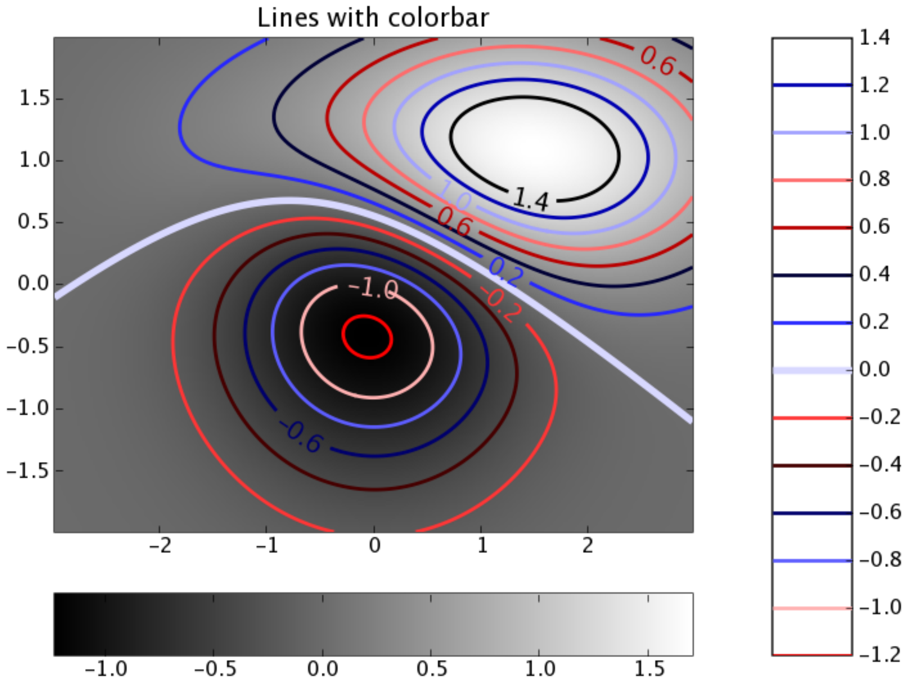


Figure 15.8: The output of `contour_demo.py`.

15.2 Visual Python

Visual Python is a Python module that makes it fairly easy to create and animate three-dimensional scenes.

Further information:

- The Visual Python home page http://vpython.org
- The Visual Python documentation (explaining all objects with all their parameters) http://vpython.org/webdoc/visual/index.html

Short videos introducing Visual Python:

- Shawn Weatherford, Jeff Polak (students of Ruth Chabay): http://www.youtube.com/vpythonvideos
- Eric Thompson: http://showmedo.com/videotutorials/series?name=pythonThompsonVPythonSeries

15.2.1 Basics, rotating and zooming

Here is an example showing how to create a red and a blue sphere at two different positions together with a flat box (`vpythondemo1.py`):

```
import visual

sphere1 = visual.sphere(pos=[0, 0, 0], color=visual.color.blue)
sphere2 = visual.sphere(pos=[5, 0, 0], color=visual.color.red, radius=2)
base = visual.box(pos=(0, -2, 0), length=8, height=0.1, width=10)
```

Once you have created such a visual python scene, you can

- rotate the scene by pressing the right mouse button and moving the mouse
- zoom in and out by pressing the middle mouse button (this could be the wheel) and moving the mouse up and down. (On some (Windows?) installations, one has to press the left and the right mouse button simultaneously and then move the move the mouse up and down to zoom.)

15.2.2 Setting the frame rate for animations

A particular strength of Visual Python is its ability to display time-dependent data:

- A very useful command is the `rate()` command which ensures that a loop is only executed at a certain frame rate. Here is an example printing exactly two "Hello World"s per second (`vpythondemo2.py`):

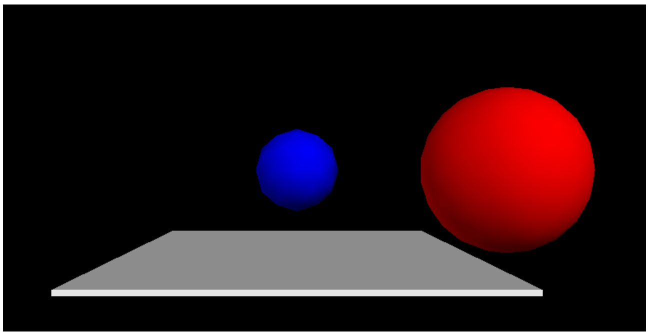


Figure 15.9: Snapshot of `vpythondemo1.py`.

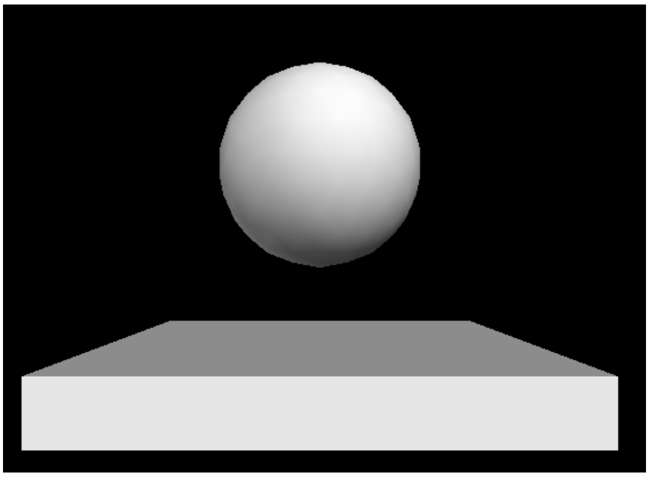


Figure 15.10: Snapshot of vpythondemo3.py.

```
import visual

for i in range(10):
    visual.rate(2)
    print("Hello World (0.5 seconds per line)")
```

- All Visual Python objects (such as the spheres and the box in the example above) have a .pos attribute which contains the position (of the centre of the object [sphere,box] or one end of the object [cylinder,helix]). Here is an example showing a sphere moving up and down (vpythondemo3.py):

```
import visual, math

ball = visual.sphere()
box = visual.box(pos=[0, -1, 0], width=4, length=4, height=0.5)

#tell visual not to automatically scale the image
visual.scene.autoscale = False

for i in range(1000):
    t = i * 0.1
    y = math.sin(t)

    #update the ball's position
    ball.pos = [0, y, 0]

    #ensure we have only 24 frames per second
    visual.rate(24)
```

15.2.3 Tracking trajectories

You can track the trajectory of an object using a "curve". The basic idea is to append positions to that curve object as demonstrated in this example (vpythondemo4.py):

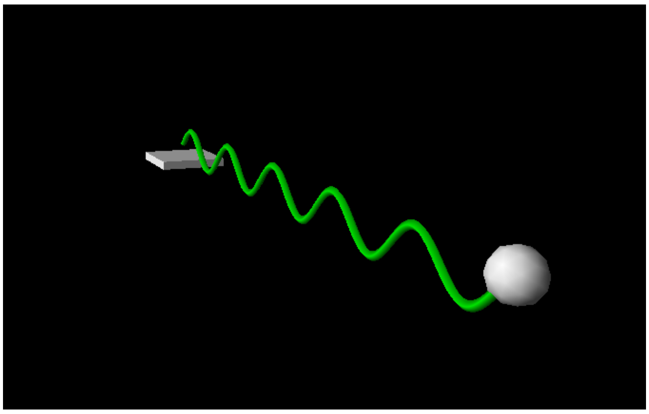


Figure 15.11: Snapshot of `vpythondemo4.py`.

```
import visual, math

ball = visual.sphere()
box = visual.box(pos=[0, -1, 0], width=4, length=4, height=0.5)
trace = visual.curve(radius=0.2, color=visual.color.green)

for i in range(1000):
    t = i * 0.1
    y = math.sin(t)

    #update the ball's position
    ball.pos = [t, y, 0]

    trace.append(ball.pos)

    #ensure we have only 24 frames per second
    visual.rate(24)
```

As with most visual Python objects, you can specify the colour of the curve (also per appended element!) and the radius.

15.2.4 Connecting objects (Cylinders, springs, . . .)

Cylinders and helices can be used to "connect" two objects. In addition to the `pos` attribute (which stores the position of one end of the object), there is also an `axis` attribute which stores the vector pointing from `pos` to the other end of the object. Here is an example showing this for a cylinder: (`vpythondemo5py`):

```
import visual, math

ball1 = visual.sphere(pos=(0, 0, 0), radius=2)
ball2 = visual.sphere(pos=(5, 0, 0), radius=2)
connection = visual.cylinder(pos=ball1.pos, \
                             axis=ball2.pos - ball1.pos)

for t in range(100):
```

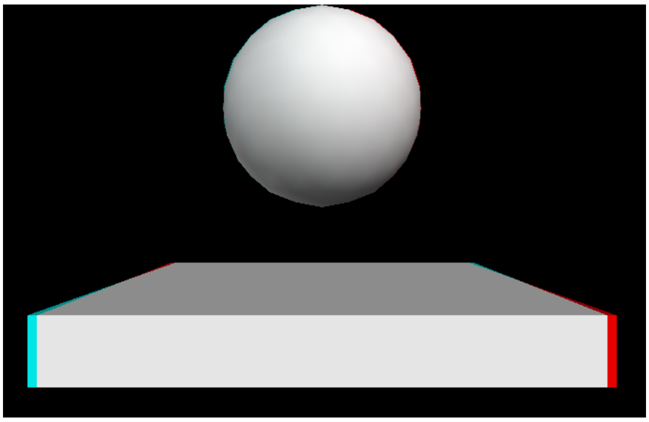


Figure 15.12: Snapshot of vpythondemo3.py in 3d mode ('redcyan'). With red/cyan glasses, this appears as a 3d image (with spatial depth).

```
    #move ball2
    ball2.pos = (-t, math.sin(t), math.cos(t))

    #keep cylinder connection between ball1 and ball2
    connection.axis = ball2.pos - ball1.pos

    visual.rate(24)
```

15.2.5 3d vision

If you have access to "anaglyphic" (*i.e.* colored) glasses (best red-cyan but red-green or red-blue works as well), then you can switch visual python into this stereo mode by adding these two lines to the beginning of your program:

```
visual.scene.stereo='redcyan'
visual.scene.stereodepth=1
```

Note the effect of the stereodepth parameter:

- stereodepth=0: 3d scene "inside" the screen (default)
- stereodepth=1: 3d scene at screen surface (this often looks best)
- stereodepth=2: 3d scene sticking out of the screen

15.3 Visualising higher dimensional data

Often, we need to understand data defined at 3d positions in space. The data itself is often a scalar field (such as temperature) or a 3d vector (such as velocity or magnetic field), or occasionally a tensor. For example for a 3d-vector field f defined in 3d-space ($\vec{f}(\vec{x})$ where $\vec{x} \in I\!R^3$ and $\vec{f}(\vec{x}) \in I\!R^3$) we could draw a 3d-arrow at every (grid) point in space. It is common for these data sets to be time dependent.

The probably most commonly used library in Science and Engineering to visualise such data sets is probably VTK, the Visualisation ToolKit (http://vtk.org). This is a substantial C++ library with interfaces to high level languages, including Python.

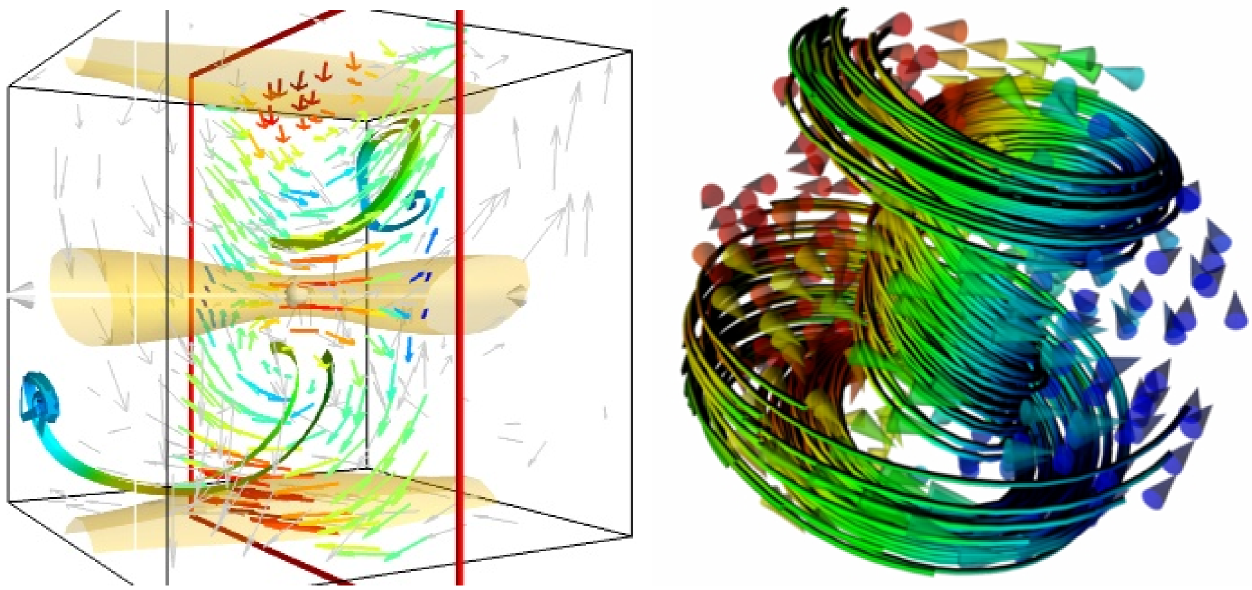
One can either call these routines directly from Python code, or write the data to disk in a format that the VTK library can read (so called vtk data files), and then use stand-alone programme such as Mayavi, ParaView and VisIt to read these data files and manipulate them (ofter with a GUI). All three of these are using the VTK library internally, and can read vtk data files.

These package is very well suited to visualise static and timedependent 2d and 3d-fields (scalar, vector and tensor fields). Two examples are shown below.

They can be used as a stand-alone executables with a GUI to visualise VTK files. It can also be scripted from a Python program, or used interactively from a Python session.

15.3.1 Mayavi, Paraview, Visit

- Mayavi Home page http://code.enthought.com/projects/mayavi/
- Paraview Home page http://paraview.org
- VisIt Home page https://wci.llnl.gov/simulation/computer-codes/visit/



Two examples from MayaVi visualisations.

15.3.2 Writing vtk files from Python (pyvtk)

A small but powerful Python library is pyvtk available at https://code.google.com/p/pyvtk/. This allows to create vtk files from Python data structures very easily.

Given a finite element mesh or a finite difference data set in Python, one can use pyvtk to write such data into files, and then use one of the visualisation applications listed above to load the vtk files and to display and investigate them.

Chapter 16

Numerical Methods using Python (scipy)

16.1 Overview

The core Python language (including the standard libraries) provide enough functionality to carry out computational research tasks. However, there are dedicated (third-party) Python libraries that provide extended functionality which

- provide numerical tools for frequently occurring tasks
- which are convenient to use
- and are more efficient in terms of CPU time and memory requirements than using the code Python functionality alone.

We list three such modules in particular:

- The numpy module provides a data type specialised for "number crunching" of vectors and matrices (this is the `array` type provided by "`numpy`" as introduced in section 14.1), and linear algebra tools.
- The `matplotlib` package (also knows as `pylab`) provides plotting and visualisation capabilities (see section 15.1) and the
- `scipy` package (SCIentific PYthon) which provides a multitude of numerical algorithms and which is introduced in this chapter.

Many of the numerical algorithms available through `scipy` and `numpy` are provided by established compiled libraries which are often written in Fortran or C. They will thus execute much faster than pure Python code (which is interpreted). As a rule of thumb, we expect compiled code to be two orders of magnitude faster than pure Python code.

You can use the help function for each numerical method to find out more about the source of the implementation.

16.2 SciPy

Scipy is built on numpy. All functionality from numpy seems to be available in scipy as well. For example, instead of

```
import numpy
x = numpy.arange(0, 10, 0.1)
y = numpy.sin(x)
```

we can therefor also use

```
import scipy as s
x = s.arange(0, 10, 0.1)
y = s.sin(x)
```

First we need to import scipy:

```
>>> import scipy
```

The scipy package provides information about its own structure when we use the help command:

```
>>> help(scipy)
```

We only show a part of the output of this command:

```
    stats        --- Statistical Functions [*]
    sparse       --- Sparse matrix [*]
    lib          --- Python wrappers to external libraries [*]
    linalg       --- Linear algebra routines [*]
    signal       --- Signal Processing Tools [*]
    misc         --- Various utilities that don't have another home.
    interpolate  --- Interpolation Tools [*]
    optimize     --- Optimization Tools [*]
    cluster      --- Vector Quantization / Kmeans [*]
    fftpack      --- Discrete Fourier Transform algorithms [*]
    io           --- Data input and output [*]
    integrate    --- Integration routines [*]
    lib.lapack   --- Wrappers to LAPACK library [*]
    special      --- Special Functions [*]
    lib.blas     --- Wrappers to BLAS library [*]
      [*] - using a package requires explicit import (see pkgload)
```

If we are looking for an algorithm to integrate a function, we might explore the `integrate` package:

```
>>> import scipy.integrate
>>> help(scipy.integrate)
```

The following sections show examples which demonstrate how to employ the algorithms provided by scipy.

16.3 Numerical integration

Scientific Python provides a number of integration routines. A general purpose tool to solve integrals I of the kind

$$I = \int_a^b f(x)\mathrm{d}x$$

is provided by the `quad()` function of the `scipy.integrate` module.

It takes as input arguments the function $f(x)$ to be integrated (the "integrand"), and the lower and upper limits a and b. It returns two values (in a tuple): the first one is the computed results and the second one is an estimation of the numerical error of that result.

Here is an example:

```
from math import cos, exp, pi
from scipy.integrate import quad


#function we want to integrate
def f(x):
    return exp(math.cos(-2 * x * pi)) + 3.2

#call quad to integrate f from -2 to 2
res, err = quad(f, -2, 2)

print("The numerical result is {:f} (+-{:g})"
    .format(res, err))
```

which produces this output:

```
The numerical result is 17.864264 (+-1.55117e-11)
```

Note that `quad()` takes optional parameters `epsabs` and `epsrel` to increase or decrease the accuracy of its computation. (Use `help(quad)` to learn more.) The default values are `epsabs=1.5e-8` and `epsrel=1.5e-8`. For the next exercise, the default values are sufficient.

16.3.1 Exercise: integrate a function

1. Using scipy's `quad` function, write a program that solves the following integral numerically: $I = \int_0^1 \cos(2\pi x)\mathrm{d}x$.

2. Find the analytical integral and compare it with the numerical solution.

3. Why is it important to have an estimate of the accuracy (or the error) of the numerical integral?

16.3.2 Exercise: plot before you integrate

It is good practice to plot the integrand function to check whether it is "well behaved" before you attempt to integrate. Singularities (i.e. x values where the $f(x)$ tends towards minus or plus infinity) or other irregular behaviour (such as $f(x) = \sin(\frac{1}{x})$ close to $x = 0$ are difficult to handle numerically.

1. Write a function with name `plotquad` which takes the same arguments as the quad command (*i.e.* f, a and b) and which (i) creates a plot of the integrand $f(x)$ and (ii) computes the integral numerically using the `quad` function. The return values should be as for the `quad` function.

16.4 Solving ordinary differential equations

To solve an ordinary differential equation of the type

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(y, t)$$

with a given $y(t_0) = y_0$ we can use **scipy**'s **odeint** function. Here is a (self explaining) example program (**useodeint.py**) to find $y(t)$ for $t \in [0, 2]$ given this differential equation:

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = -2yt \quad \text{with} \quad y(0) = 1.$$

```
from scipy.integrate import odeint
import numpy as N

def f(y, t):
    """this is the rhs of the ODE to integrate, i.e. dy/dt=f(y,t)"""
    return -2 * y * t

y0 = 1              # initial value
a = 0               # integration limits for t
b = 2

t = N.arange(a, b, 0.01)  # values of t for
                          # which we require
                          # the solution y(t)
y = odeint(f, y0, t)  # actual computation of y(t)

import pylab          # plotting of results
pylab.plot(t, y)
pylab.xlabel('t'); pylab.ylabel('y(t)')
pylab.show()
```

which produces the graph shown in figure 16.1.

The **odeint** command takes a number of optional parameters to change the default error tolerance of the integration (and to trigger the production of extra debugging output). Use the help command to explore these:

```
>>> help(scipy.integrate.odeint)
```

16.4.1 Exercise: using `odeint`

1. Open a new file with name `testodeint.py` file in the IDLE editor.

2. Write a program that computes the solution $y(t)$ of this ODE using the `odeint` algorithm:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\exp(-t)(10\sin(10t) + \cos(10t)) \qquad (16.1)$$

   from $t = 0$ to $t = 10$. The initial value is $y(0) = 1$.

3. You should display the solution graphically at points $t = 0, t = 0.01, t = 0.02, \ldots, t = 9.99, t = 10$.

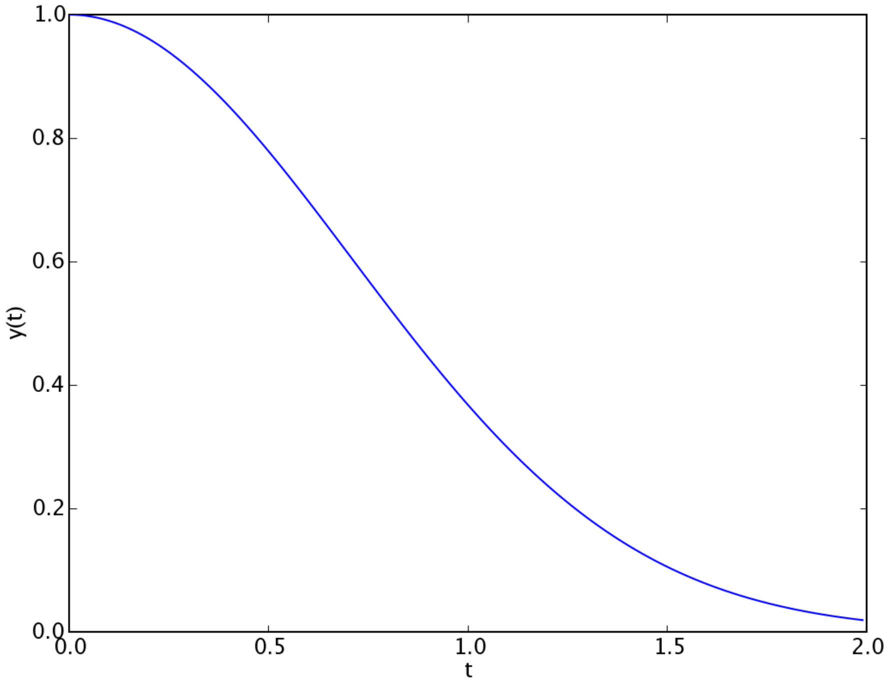
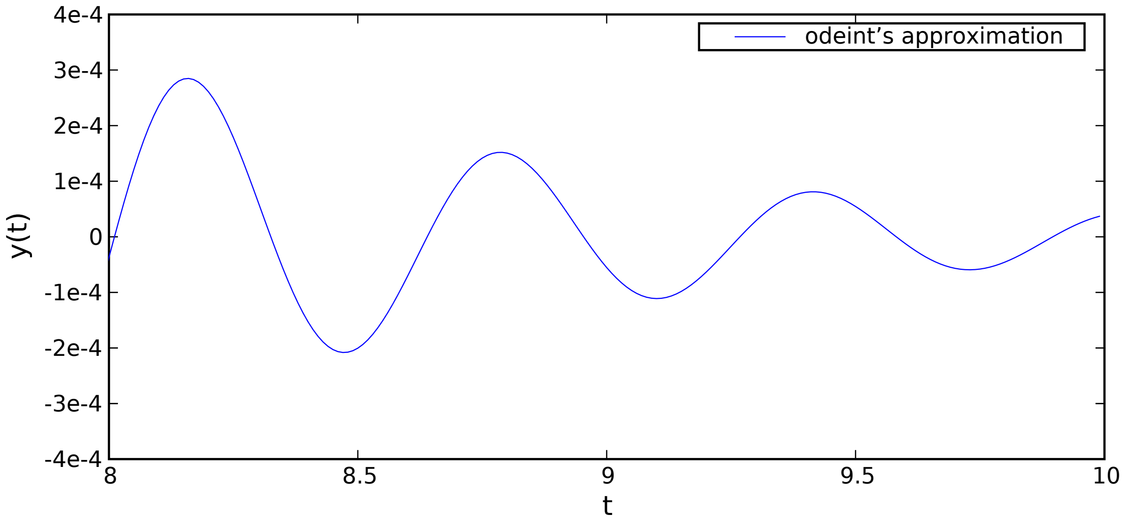


Figure 16.1: The output of `useodeint.py`.

Hint: a part of the solution $y(t)$ is shown in the figure below.



16.5 Root finding

If you try to find a x such that

$$f(x) = 0,$$

then this is called *root finding*. Note that problems like $g(x) = h(x)$ fall in this category as you can rewrite them as $f(x) = g(x) - h(x) = 0$.

A number of root finding tools are available in `scipy`'s `optimize` module.

16.5.1 Root finding using the bisection method

First we introduce the `bisect` algorithm which is (i) robust and (ii) slow but conceptually very simple.

Suppose we need to compute the roots of $f(x) = x^3 - 2x^2$. This function has a (double) root at $x = 0$ (this is trivial to see) and another root which is located between $x = 1.5$ (where $f(1.5) = -1.125$) and $x = 3$ (where $f(3) = 9$). It is pretty straightforward to see that this other root is located at $x = 2$. Here is a program that determines this root numerically:

```
from scipy.optimize import bisect

def f(x):
    """returns f(x)=x^3-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x ** 3 - 2 * x ** 2

#main program starts here
x = bisect(f, 1.5, 3, xtol=1e-6)

print "The root x is approximately x=%14.12g,\n"\
      "the error is less than 1e-6." % (x)
print "The exact error is %g." % (2 - x)
```

This produces the following output:

```
The root x is approximately x= 2.00000023842,
the error is less than 1e-6.
The exact error is -2.38419e-07.
```

The `bisect()` method takes three compulsory arguments: (i) the function $f(x)$, (ii) a lower limit a (for which we have chosen 1.5 in our example) and (ii) an upper limit b (for which we have chosen 3). The optional parameter `xtol` determines the maximum error of the method.

One of the requirements of the bisection method is that the interval $[a, b]$ has to be chosen such that the function is either positive at a and negative at b, or that the function is negative at a and postive at b. In other words: a and b have to enclose a root.

16.5.2 Exercise: root finding using the bisect method

1. Write a program with name `sqrttwo.py` to determine an approximation of $\sqrt{2}$ by finding a root x of the function $f(x) = 2 - x^2$ using the bisection algorithm. Choose a tolerance for the approximation of the root of 10^{-8}.

2. Document your choice of the initial bracket $[a, b]$ for the root: which values have you chosen for a and for b and why?

3. Study the results:

- Which value for the root x does the bisection algorithm return?
- Compute the value of $\sqrt{2}$ using `math.sqrt(2)` and compare this with the approximation of the root. How big is the absolute error of x? How does this compare with `xtol`?

16.5.3 Root finding using the `fsolve` funcion

A (often) better (in the sense of "more efficient") algorithm than the bisection algorithm is implemented in the general purpose `fsolve()` function for root finding of (multidimensional) functions. This algorithm needs only one starting point close to the suspected location of the root (but is not garanteed to converge).

Here is an example:

```
from scipy.optimize import fsolve

def f(x):
    return x ** 3 - 2 * x ** 2

x = fsolve(f, 3)           # one root is at x=2.0

print "The root x is approximately x=%21.19g" % x
print "The exact error is %g." % (2 - x)
```

Which produces this output:

```
The root x is approximately x= 2.000000000000006661
The exact error is -6.66134e-15.
```

The return value[1] of `fsolve` is a numpy array of length n for a root finding problem with n variables. In the example above, we have $n = 1$.

16.6 Interpolation

Given a set of N points (x_i, y_i) with $i = 1, 2, \ldots N$, we sometimes need a function $\hat{f}(x)$ which returns $y_i = f(x_i)$ where $x == x_i$, and which in addition provides some interpolation of the data (x_i, y_i) for all x.

The function `y0 = scipy.interpolate.interp1d(x,y,kind='nearest')` does this interpolation based on splines of varying order. Note that the function `interp1d` returns *a function* `y0` which will then interpolate the x-y data for any given x when called as $y0(x)$.

The code below demonstrates this, and we show the different interpolation kinds in figure 16.3.

```
import numpy as np
import scipy.interpolate
import pylab

def create_data(n):
    """Given an integer n, returns n data points
    x and values y as a numpy.array."""
    xmax = 5.
    x = np.linspace(0, xmax, n)
    y = - x**2
```

[1]Historical note: this has changed from scipy version 0.7 to 0.8. Before 0.8, the return value was a float if a one-dimensional problem was to solve.

```
    #make x-data somewhat irregular
    y += 1.5 * np.random.normal(size=len(x))
    return x, y

#main program
n = 10
x, y = create_data(n)

#use finer and regular mesh for plot
xfine = np.linspace(0.1, 4.9, n * 100)
#interpolate with piecewise constant function (p=0)
y0 = scipy.interpolate.interp1d(x, y, kind='nearest')
#interpolate with piecewise linear func (p=1)
y1 = scipy.interpolate.interp1d(x, y, kind='linear')
#interpolate with piecewise constant func (p=2)
y2 = scipy.interpolate.interp1d(x, y, kind='quadratic')

pylab.plot(x, y, 'o', label='data point')
pylab.plot(xfine, y0(xfine), label='nearest')
pylab.plot(xfine, y1(xfine), label='linear')
pylab.plot(xfine, y2(xfine), label='cubic')
pylab.legend()
pylab.xlabel('x')
pylab.savefig('interpolate.pdf')
pylab.show()
```

16.7 Curve fitting

We have already seen in Sect. 16.7 that we can fit polynomial functions through a data set using the `numpy.polyfit` function. Here, we introduce a more generic curve fitting algorithm.

Scipy provides a somewhat generic function (based on the Levenburg-Marquardt algorithm)through `scipy.optimize.curve_fit` to fit a given (Python) function to a given data set. The assumption is that we have been given a set of data with points $x_1, x_2, \ldots x_N$ and with corresponding function values y_i and a dependence of y_i on x_i such that $y_i = f(x_i, \vec{p})$. We want to determine the parameter vector $\vec{p} = (p_1, p_2, \ldots, p_k)$ so that r, the sum of the residuals, is as small as possible:

$$r = \sum_{i=1}^{N} (y_i - f(x_i, \vec{p}))^2 . \tag{16.2}$$

Curve fitting is of particular use if the data is noisy: for a given x_i and $y_i = f(x_i, \vec{p})$ we have a (unknown) error term ϵ_i so that $y_i = f(x_i, \vec{p}) + \epsilon_i$.

We use the following example to clarify this:

$$f(x, \vec{p}) = a \exp(-bx) + c, \quad \text{i.e.} \quad \vec{p} = \mathtt{a}, \mathtt{b}, \mathtt{c}. \tag{16.3}$$

```
import numpy as np
from scipy.optimize import curve_fit
```

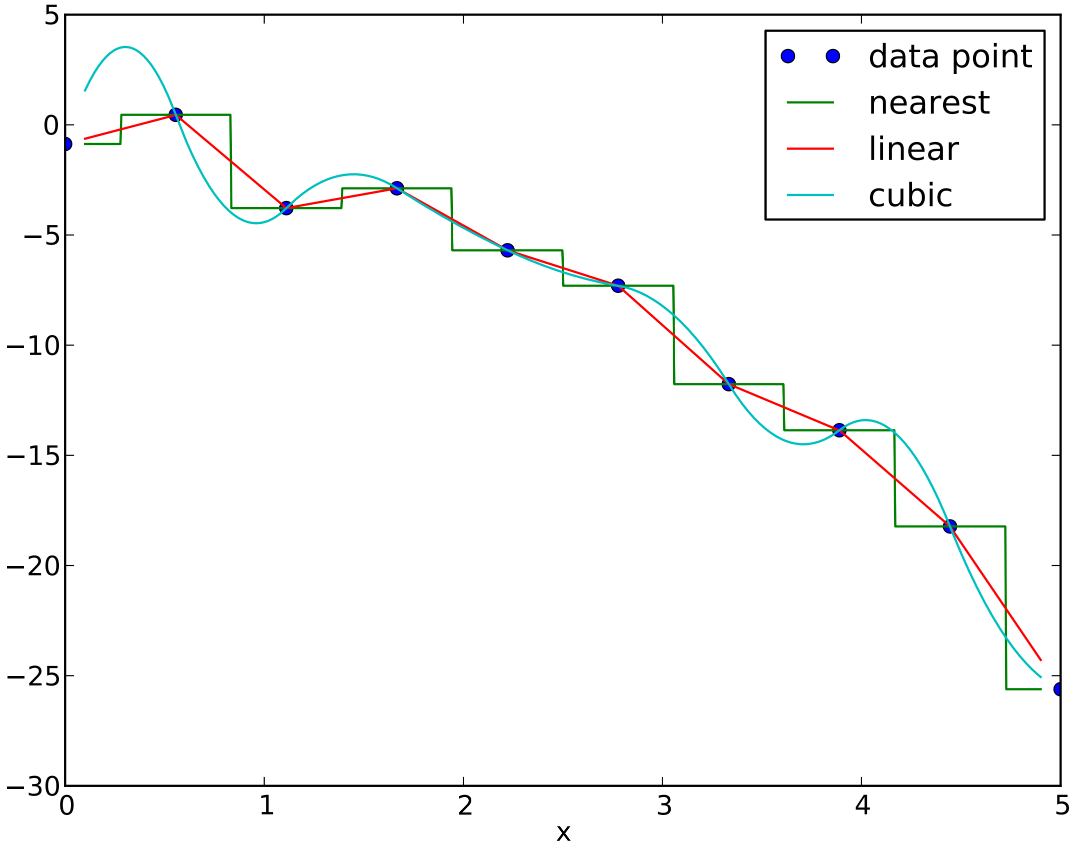


Figure 16.2: Blue dots are interpolated by piecewise polynomial of order 0 ("nearest"), order 1("linear") and order 3 ("cubic").

```
def f(x, a, b, c):
    """Fit function y=f(x,p) with parameters p=(a,b,c). """
    return a * np.exp(- b * x) + c

#create fake data
x = np.linspace(0, 4, 50)
y = f(x, a=2.5, b=1.3, c=0.5)
#add noise
yi = y + 0.2 * np.random.normal(size=len(x))

#call curve fit function
popt, pcov = curve_fit(f, x, yi)
a, b, c = popt
print "Optimal parameters are a=%g, b=%g, and c=%g" % (a, b, c)

#plotting
import pylab
yfitted = f(x, *popt)   # equivalent to f(x, popt[0], popt[1], popt[2])
pylab.plot(x, yi, 'o', label='data $y_i$')
pylab.plot(x, yfitted, '-', label='fit $f(x_i)$')
pylab.xlabel('x')
pylab.legend()
pylab.savefig('curvefit.png')
```

Plotting the noisy data together with the fit, we get the plot in figure 16.3.

Note that in the source code above we define the fitting function $y = f(x)$ through Python code. We can thus fit (nearly) arbitrary functions using the `curve_fit` method.

The `curve_fit` function returns a tuple `popt, pcov`. The first entry `popt` contains a tuple of the OPTimal Parameters (in the sense that these minimise equation (16.2). The second entry contains the covariance matrix for all parameters. The diagonals provide the variance of the parameter estimations.

For the curve fitting process to work, the Levenburg-Marquardt algorithm needs to start the fitting process with initial guesses for the final parameters. If these are not specified (as in the example above), the value "1.0" is used for the initial guess.

If the algorithm fails to fit a function to data (even though the function describes the data reasonably), we need to give the algorithm better estimates for the initial parameters. For the example shown above, we could give the estimates to the `curve_fit` function by changing the line

```
popt, pcov = curve_fit(f, x, yi)
```

to

```
popt, pcov = curve_fit(f, x, yi, p0=(2,1,0.6))
```

if our initial guesses would be $a = 2, b = 1$ and $c = 0.6$. Once we take the algorithm "roughly in the right area" in parameter space, the fitting usually works well.

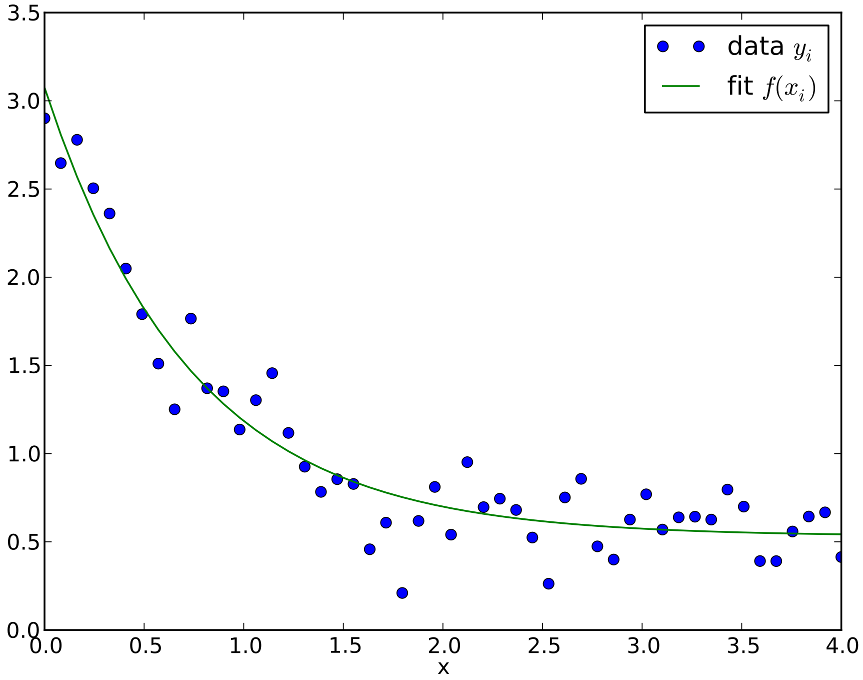


Figure 16.3: Noisy input data y_i and fitting result $f(x, \vec{p})$ (see (16.3) . The particular fitted parameters for this run a=2.3669, b=1.06078, and c=0.464289.

16.8 Fourier transforms

In the next example, we create a signal as a superposition of a 50 Hz and 70 Hz sine wave (with a slight phase shift between them). We then Fourier transform the signal and plot the absolute value of the (complex) discrete Fourier transform coefficients against frequency, and expect to see peaks at 50Hz and 70Hz.

```
import scipy
import matplotlib.pyplot as plt
pi = scipy.pi

signal_length = 0.5    #[seconds]
sample_rate=500        #sampling rate [Hz]
dt = 1./sample_rate    #time between two samples [s]

df = 1/signal_length   #frequency between points in
                       #in frequency domain [Hz]
t=scipy.arange(0,signal_length,dt) #the time vector
n_t=len(t)             #length of time vector

#create signal
y=scipy.sin(2*pi*50*t)+scipy.sin(2*pi*70*t+pi/4)

#compute fourier transform
f=scipy.fft(y)

#work out meaningful frequencies in fourier transform
freqs=df*scipy.arange(0,(n_t-1)/2.,dtype='d') #d=double precision float
n_freq=len(freqs)

#plot input data y against time
plt.subplot(2,1,1)
plt.plot(t,y,label='input data')
plt.xlabel('time [s]')
plt.ylabel('signal')

#plot frequency spectrum
plt.subplot(2,1,2)
plt.plot(freqs,abs(f[0:n_freq]),
         label='abs(fourier transform)')
plt.xlabel('frequency [Hz]')
plt.ylabel('abs(DFT(signal))')

#save plot to disk
plt.savefig('fft1.pdf')
plt.show()             #and display plot on screen
```

The resulting plot is shown in figure 16.4.

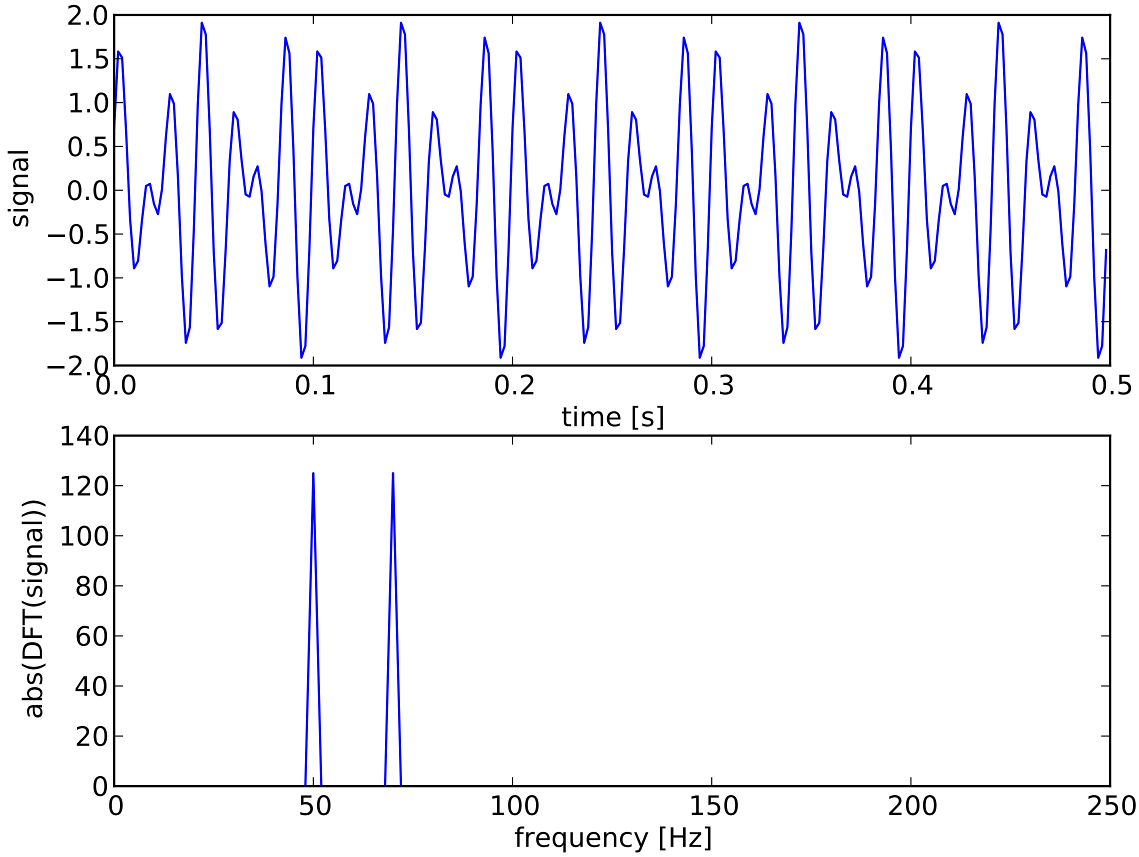


Figure 16.4: The output of `fft1.py`. The lower plot shows the discrete Fourier transform computed from the data shown in the upper plot.

16.9 Optimisation

Often we need to find the maximum or minimum of a particular function $f(x)$ where f is a scalar function but x could be a vector. Typical applications are the minimisation of entities such as cost, risk and error, or the maximisation of productivity, efficiency and profit. Optimisation routines typically provide a method to minimise a given function: if we need to maximise $f(x)$ we create a new function $g(x)$ that reverses the sign of f, i.e. $g(x) = -f(x)$ and we minimise $g(x)$.

Below, we provide an example showing (i) the definition of the test function and (ii) the call of the `scipy.optimize.fmin` function which takes as argument a function f to minimise and an initial value x_0 from which to start the search for the minimum, and which returns the value of x for which $f(x)$ is (locally) minimised. Typically, the search for the minimum is a local search, i.e. the algorithm follows the local gradient. We repeat the search for the minimum for two values ($x_0 = 1.0$ and $x_0 = 2.0$, respectively) to demonstrate that depending on the starting value we may find different minimar of the function f.

The majority of the commands (after the two calls to `fmin`) in the file `fmin1.py` creates the plot of the function, the start points for the searches and the minima obtained:

```
from scipy import arange, cos, exp
from scipy.optimize import fmin
import pylab

def f(x):
    return cos(x) - 3 * exp( -(x - 0.2) ** 2)

# find minima of f(x),
# starting from 1.0 and 2.0 respectively
minimum1 = fmin(f, 1.0)
print "Start search at x=1., minimum is",minimum1
minimum2 = fmin(f, 2.0)
print "Start search at x=2., minimum is",minimum2

# plot function
x = arange(-10, 10, 0.1)
y = f(x)
pylab.plot(x, y, label='$\cos(x)-3e^{-(x-0.2)^2}$')
pylab.xlabel('x')
pylab.grid()
pylab.axis([-5, 5, -2.2, 0.5])

# add minimum1 to plot
pylab.plot(minimum1, f(minimum1), 'vr',
           label='minimum 1')
# add start1 to plot
pylab.plot(1.0, f(1.0), 'or', label='start 1')

#add minimum2 to plot
pylab.plot(minimum2,f(minimum2),'vg',\
           label='minimum 2')
#add start2 to plot
pylab.plot(2.0,f(2.0),'og',label='start 2')
```

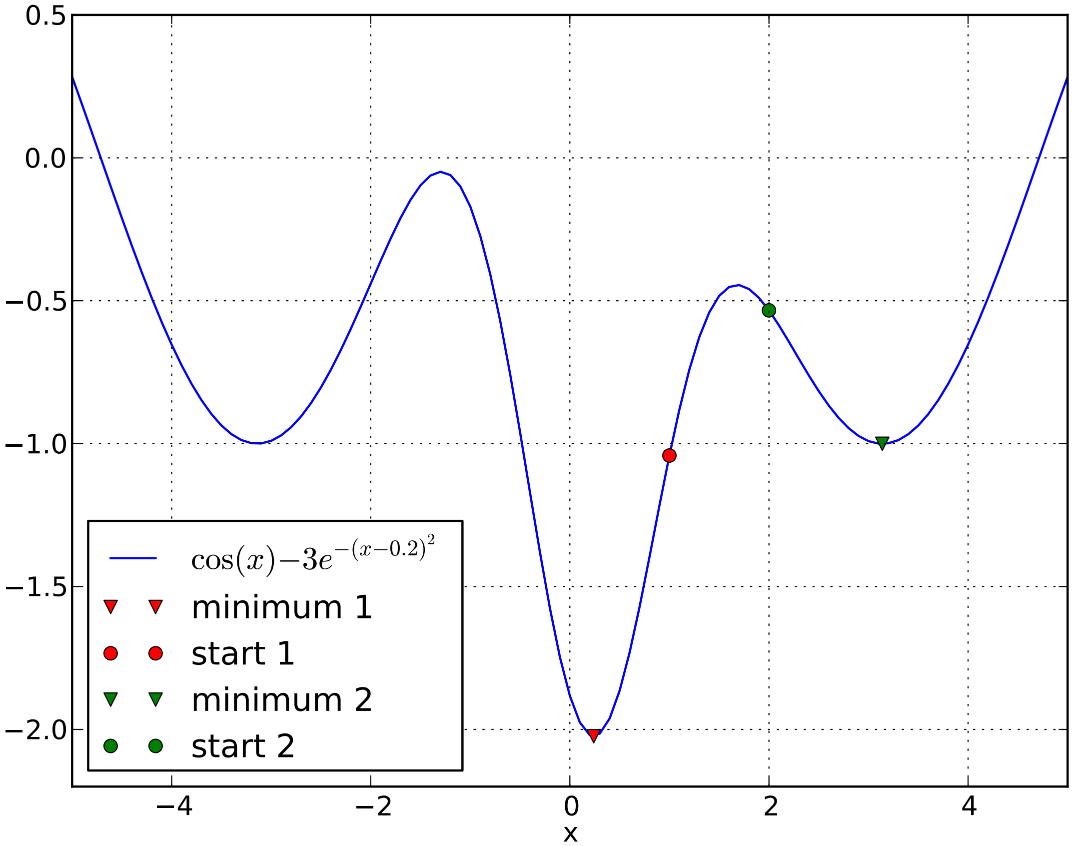


Figure 16.5: The output of `fmin1.py`. The minimisation routine `fmin` finds the next local minimum starting from a user-provided initial position.

```
pylab.legend(loc='lower left')
pylab.savefig('fmin1.pdf')
pylab.show()
```

The resulting plot is shown in figure 16.5.

Calling the `fmin` function, will produce some diagnostic output which can be seen in the following capture of the stdout from running the `fmin1.py` file:

```
Optimization terminated successfully.
         Current function value: -2.023866
         Iterations: 16
         Function evaluations: 32
Start search at x=1., minimum is [ 0.23964844]
Optimization terminated successfully.
         Current function value: -1.000529
         Iterations: 16
         Function evaluations: 32
Start search at x=2., minimum is [ 3.13847656]
```

Return value of `fmin` Note that the return value from the `fmin` function is a numpy `array` which – for the example above – contains only one number as we have only one parameter (here x) to vary. In general, `fmin` can be used to find the minimum in a higher-dimensional parameter space if there are several parameters. In that case, the numpy array would contain those parameters that minimise the objective function. The objective function $f(x)$ has to return a scalar even if there are more parameters, i.e. even if x is a vector as in $f(\mathbf{x})$.

16.10 Other numerical methods

Scientific Python and Numpy provide access to a large number of other numerical algorithms including function interpolation, Fourier transforms, optimisation, special functions (such as Bessel functions), signal processing and filters, random number generation, and more. Start to explore scipy's and numpy's capabilities using the help function and the documentation provided on the web.

16.11 scipy.io: Scipy-input output

Scipy provides routines to read and write Matlab mat files. Here is an example where we create a Matlab compatible file storing a (1x11) matrix, and then read this data into a numpy array from Python using the scipy Input-Output library:

First we create a mat file in Octave (Octave is [mostly] compatible with Matlab)::

```
octave:1> a=-1:0.5:4
a =
Columns 1 through 6:
   -1.0000   -0.5000    0.0000    0.5000    1.0000    1.5000
Columns 7 through 11:
   2.0000    2.5000    3.0000    3.5000    4.0000
octave:2> save -6 octave_a.mat a       %save as version 6
```

Then we load this array within python:

```
>>> from scipy.io import loadmat
>>> mat_contents = loadmat('octave_a.mat')
{'__globals__': [],
 '__header__': 'MATLAB 5.0 MAT-file, Octave 3.2, 2011-09-16 22:13:21',
 '__version__': '1.0',
 'a': array([[-1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,\
 3. ,  3.5,  4. ]])}
>>> mat_contents['a']
array([[-1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  \
    3. ,  3.5,  4. ]])
```

The function loadmat returns a dictionary: the key for each item in the dictionary is a string which is the name of that array when it was saved in Matlab. The key is the actual array.

A Matlab matrix file can hold several arrays. Each of those is presented by one key-value pair in the dictionary.

Let's save two arrays from Python to demonstrate that:

```
import scipy.io
import numpy as np

#create two numpy arrays
a = np.linspace(0, 50, 11)
b = np.ones((4, 4))

#save as mat-file
#create dictionary for savemat
tmp_d = {'a': a,
         'b': b}
```

```
scipy.io.savemat('data.mat', tmp_d)
```

This program creates the file `data.mat`, which we can subsequently read using Matlab or here Octave:

```

HAL47:code fangohr$ octave
GNU Octave, version 3.2.4
Copyright (C) 2009 John W. Eaton and others.
<snip>

octave:1> whos
Variables in the current scope:

  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       ans         1x11                        92  cell

Total is 11 elements using 92 bytes

octave:2> load data.mat
octave:3> whos
Variables in the current scope:

  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       a          11x1                         88  double
       ans         1x11                        92  cell
       b           4x4                        128  double

Total is 38 elements using 308 bytes

octave:4> a
a =

    0
    5
   10
   15
   20
   25
   30
   35
   40
   45
   50

octave:5> b
b =

   1   1   1   1
```

```
    1     1     1     1
    1     1     1     1
    1     1     1     1
```

Note that there are other functions to read from and write to in formats as used by IDL, Netcdf and other formats in `scipy.io`.

More → see Scipy tutorial.

Chapter 17

Where to go from here?

Learning a programming language is the first step towards becoming a computationalists who advances science and engineering through computational modelling and simulation.

We list some additional skills that can be very beneficial for day-to-day computational science work, but is of course not exhaustive.

17.1 Advanced programming

This text has put emphasis on providing a robust foundation in terms of programming, covering control flow, data structures and elements from function and procedural programming. We have not touch Object Orientation in great detail, nor have we discussed some of Python's more advanced features such as iterators, and decorators, for example.

17.2 Compiled programming language

When performance starts to be the highest priority, we may need to use compiled code, and likely embed this in a Python code to carry out the computational that are the performance bottle neck.

Fortran, C and C++ are sensible choices here; maybe Julia in the near future.

We also need to learn how to integrate the compiled code with Python using tools such as Cython, Boost, Ctypes and Swig.

17.3 Testing

Good coding is supported by a range of unit and system tests that can be run routinely to check that the code works correctly. Tools such as doctest, nose and pytest are invaluable, and we should learn at least how to use pytest (or nose).

17.4 Simulation models

A number of standard simulation tools such as Monte Carlo, Molecular Dynamics, lattice based models, agents, finite difference and finite element models are commonly used to solve particular simulation challenges – it is useful to have at least a broad overview of these.

17.5 Software engineering for research codes

Research codes bring particular challenges: the requirements may change during the run time of the project, we need great flexibility yet reproducibility. A number of techniques are available to support effectively.

17.6 Data and visualisation

Dealing with large amounts of data, processing and visualising it can be a challenge. Fundamental knowledge of database design, 3d visualisation and modern data processing tools such as the Pandas Python package help with this.

17.7 Version control

Using a version control tool, such as git or mercurial, should be a standard approach and improves code writing effectiveness significantly, helps with working in teams, and - maybe most importantly - supports reproducibility of computational results.

17.8 Parallel execution

Parallel execution of code is a way to make it run orders of magnitude faster. This could be using MPI for inter-node communication or OpenMP for intra-node parallelisation or a hybrid mode bringing both together.

The recent rise of GPU computing provides yet another avenue of parallelisation, and so do the many-core chips such as the Intel Phi.

Acknowledgements

Thanks go to

- Marc Molinari for carefully proof reading this manuscript around 2007.
- Neil O'Brien for contributing to the SymPy section.
- Jacek Generowicz for introducing me to Python in the last millennium, and for kindly sharing countless ideas from his Python course.
- EPSRC for partial support of this work (GR/T09156/01 and EP/G03690X/1).
- Students and other readers who have provided feedback and pointed out typos and errors etc.


```
changeset:   418:321f2967175e
tag:         tip
user:        Hans Fangohr [iota] <fangohr@soton.ac.uk>
date:        Mon Sep 07 09:44:40 2015 +0100
summary:     minor improvements from proofreading
```